1616: User Tutorial

Version 4.091

August 1993

Beginner's Tutorial Manual
Applix 1616 microcomputer project
Applix pty limited

1616 User Tutorial Manual

Even though Applix has tested the software and reviewed the documentation, Applix makes no warranty or representation, either express or implied, with respect to software, its quality, performance, merchantability, or fitness for a particular purpose. As a result this software is sold "as is," and you the purchaser are assuming the entire risk as to its quality and performance.

In no event will Applix be liable for direct, indirect, special, incidental, or consequential damages resulting from any defect in the software or its documentation.

The original version of this manual was written by Andrew Morton
Additional introductory and tutorial material by Eric Lindsay
Editorial and design consultant: Jean Hollis Weber

Comments about this manual or the software it describes should be sent to:

Applix Pty Limited
Lot 1, Kent Street,
Yerrinbool, 2575
N.S.W. Australia
(048) 839 372
Private Applix BBS numbers include Colin McCormack (02) 543 8213, SSM (02) 554 3114, Trantor (02) 718 6996 and PPT (02) 544 1060 (on ringback - let the phone ring twice, then phone back within 30 seconds).
User Group meetings are generally held on the second Saturday of each month, from about 1 p.m. until evening. All computer enthusiasts are welcome, as are new visitors. Come in and meet the designer of the Applix system.

ISBN 0 947341 00 5

MC68000®™ is a trademark of Motorola Inc.

1
Introduction

This manual is an introduction to the inbuilt operating system (1616/OS) for the Applix 1616 computer. Many commands described in this manual, including multitasking, are available on every Applix 1616 the moment you apply power, because they are built into the eproms on the Applix motherboard.

Unlike other computers, you do not need disk drives to use an Applix. The Applix 1616 will start whether a disk drive is supplied or not, however some activities are rather restricted. We recommend that all experienced users obtain floppy disk drives (and possibly hard drives as well), to fully enjoy using the system. This manual does not specifically cover the setting up of Applix disk drives.

This chapter briefly describes the many manuals and programs available for the Applix 1616. It indicates the intended audience for the computer, the uses to which it might be put, and the facilities available. The operating system is briefly described, together with suggestions about standards, plus how to enhance and upgrade the system. Finally, the typographic and keyboard conventions used throughout these manuals are described.

The Applix 1616 is designed to be helpful and easy to use for programmers, rather than neccessarily easier for end users of programs. This is an important distinction that makes the Applix 1616 unique among home and educational computers, because it enormously increases the power available to the new programmer.

The manual

This manual includes brief introductions on some of the concepts underlying 1616/OS, and includes some examples of their use. It also includes tutorials on some of more complex utilities, such as the extensive command line editor, and the full screen editor.

A separate reference section towards the end of this manual lists by category all the inbuilt commands that appear in the 1616/OS operating system program. The many programs available on the Applix User Disk are described in a separate manual.

The original version was written by Andrew Morton for builders of his Applix 1616 kit computer. It now includes additional tutorial material, and introduces some concepts with which a new computer owner may not be familiar.

If you have no background in computers, you will find it helpful to read some introductory books while you learn about your new computer. We recommend the cheap *Understanding ...* range devised by Texas Instruments, and sometimes available from Tandy or Radio Shack. A list and description of these is contained in the *Applix Hardware and Construction Manual*, although I must warn that these books may now be difficult to locate.

Those readers familiar with home computers, and especially MS-DOS or UNIX, may use the brief summary of some concepts and commands in Appendix A, to get you started quickly. We do however advise checking the contents page, and index pages, as some specialised programs are considerably different to those with which you may be familiar. In particular, 1616/OS is far more powerful, and includes many more utilities, than any normal home computer. If you ignore the facilities described in these manuals, you will cripple your operation of the system.

The audience

We assume that one of your motives in buying an Applix 1616 is to learn more about computers, or to accomplish some task for which normal business systems are unsuitable. Application programs are not readily available for the Applix 1616, and so the 1616 is not an appropriate choice for someone wishing to run standard business applications such as a popular spreadsheet.

To use the Applix 1616, you must first learn to use its inbuilt commands. These commands allow you to control the disk drive, see the contents of files, print out material, and do all the 'computer housekeeping' that is required by any computer. You use these commands simply by typing their name, followed by optional *parameters*, which modify how they work.

All computer systems require you to use commands to control them, and the sooner you learn these commands, the more use the system will be. In some systems, particularly in business, some of the commands are hidden by menus, from which you select commands. However, you often find that you learn no more than what the menu provides.

Other computers, such as the Apple Macintosh, Atari ST and Commodore Amiga, use windows, icons, and a mouse pointer (the *wimp* interface), to insulate you from actually typing commands. Although much easier to use, such a system can also insulate you from the full power of your computer. However, at some level, the equivalent commands exist in all computers.

In the Applix, control comes from you knowing the commands, and typing them correctly. This manual provides the first steps in this process.

A command is simply an inbuilt program, intended to do something that is of use, either to you, or to the computer. Some commands do simple, very understandable jobs (like tell the time), while others have purposes that will not be clear until you use them extensively yourself. In the Applix, commands are intended to work together, in a manner similar to UNIX. This allows you to build your own commands, even before you learn programming.

Contents

Amongst other things, this manual includes the following topics:

- Background information.
- How to start the computer.

- How to use the keyboard, and various special keys that can save you effort.
- How to use the inbuilt line editor to avoid re-typing material.
- What we mean by computer terms such as files, disks, directories, floppy disks, hard disks, hierarchical filing systems, and mass storage devices.
- How to use the inbuilt commands.
- Writing '`shell`' programs to avoid retyping involved or lengthy groups of commands.
- Re-directing information to and from different parts of the computer, and to and from the outside world.
- Using the screen editor as a simple word processor and for general text editing.
- Using commands to save files on disk or tape, and read them again.
- Using commands to print files on paper, or send them to another computer.
- Using the inbuilt monitor to examine and change the contents of memory.
- Using the 1616 as a terminal to another computer.
- Using inbuilt application programs to tell the time, do arithmetic, convert numerical expressions, obtain help, etc.

This is an introductory users manual only; it does not mention programming, nor does it cover the powerful 'system calls' available in the 1616. It does not attempt to explain how to use the program languages available for the Applix 1616. Each language has its own manual, listing the commands available. In addition, when you learn programming, you will need a separate manual about the programming language in question. Use of the many powerful lower level 'system calls' is covered in the *Programmer's Manual*, and the *Technical Reference Manual*. The hardware and electronic design of the 1616 is covered in the *Hardware Manual*.

Other manuals and programs

A wide variety of manuals are available for the Applix 1616 system. Extra copies, updates, or copies for evaluation, are available at $10 per manual at any time. The manuals available include:

- Users Disk Manual: A program by program description of some of the many handy utilities available on the User Disk or via the Applix bulletin board.
- Hardware and Construction Manual: Complete details of how to build the 1616, including parts lists, step by step instructions, schematics, design overview, detailed hardware description, connector pinouts, switch and link settings, cable connections, test and trouble-shooting procedures.
- Disk Co-processor Card: Parts list, step by step construction instructions, design overview and hardware description, explanation of the software, connector pinouts, switch and link settings, and schematics.

- Hard Disk Manual: Explanation of the action of the SCSI port, the hard disk, and how to connect them. Explanation of the hard disk software, and additional utility programs for formatting, testing and using hard disks.
- Programmer's Manual: Short explanation of programming the 1616, how to use system calls, a complete list of system calls, including file control, character I/O, video output, and graphics control.
- Technical Reference Manual: Using relocatable code, writing memory resident drivers, additional material on system calls, including character device drivers, disk device drivers, file systems, mouse, multitasking, etc.
- Assembler Users Manual: Brief overview of using 68000 assembler, assembler directives, structured programming, macro and conditional assembly, linker errors.
- Shareware Manuals: An expanding set of manuals, on disk, produced by the Applix Users Group. These explain how to use a number of programs available on the first 30 shareware disks from Applix, all of which include source code.

A number of additional programs are available at extra cost, together with manuals. These include:

- BASIC Interpreter: Lists all commands and functions available in SSBASIC, with additional information on calling machine code, and using files. Converted by Andrew Morton, of Applix, $69.
- Hi-Tech C Compiler: Using C, features, standard libraries, style, pointers, C reference, linker, libraries. A standard K&R C, from a well-known Australian company, at $250.
- C Tutorial: A simple introduction to the C language, with over 50 programs on disk. Many can be used with the shareware *Superscript* mini C interpreter.
- Forth, fully extended version of the one made famous in Dr Dobbs, ported to Applix by Peter Fletcher. Documented on disk, plus reprinted manual, including full source code, $89.
- CP/M (ZCPR3 and ZRDOS) ported by Conal Walsh to run on the 8 MHz Z80 CPU on the Applix disk controller card. Very fast version, with full colour display. Includes disk controller upgrade to 64k. Short manual and disks, and now includes hard disk support. Reads MicroBee and other CP/M disks, and costs less than $140.
- Minix, Andrew Tanenbaum's version of the UNIX V7 operating system, ported by Colin McCormack of Tangled Web Software. Includes over 100 utility programs, manual, and four disks which include the complete source code, for round $200. A fine way to teach operating system design.
- Applix Utilities #1: A variety of UNIX style utility programs, about 30 in all, for text formatting, recovering files, sorting, searching etc., two disks worth, including source code, all for $29.95.

- Dr Doc: The Applix document editor, a WordStar-like full screen editor compatible with the inbuilt editor. Includes right and left justification of text, support for on screen display of italics, bold, underline, subscript and superscript. Provides headers and footers, page numbering, date, on-line help, etc. Easy printer driver changes. Printed manual, but no source. Only $29.95.
- Mgr: A free program by Stephen Uhler, from Bellcore. Ported from Sun workstations, this is a full windowing system, with mouse support. We advise video modification for 960 by 512 display. Run Dr Doc editor with cut and paste, slide bars, open, close and resize windows, etc. $29.95 for video modification needed to run it (two PALs, includes free mgr executables on two disks). Four disks of source code available, if required. Printed manual also available, at the cost of photocopying.

The Applix 1616

Welcome to the world of the Applix 1616 User. The 1616 is designed as a powerful, but **easy to program**, general purpose computer.

It is particularly suited to education (and self instruction), especially for learning low level (assembler or Forth) programming, and for interfacing to external devices. It is suitable for higher level language programming, in Basic and the C language. Due to the accessibility of the operating system, and its relatively advanced nature, it can be used in courses about operating system design. In addition, Andrew Tanenbaum's famous Minix operating system, designed especially for academic courses on operating systems, has been ported by Colin McCormack of Tangled Web Software, and is available for round $200.

The Applix 1616 is suited to light industrial and home control use, or controlling and recording scientific instruments, thanks to the extensive inbuilt communication and input output facilities, and its expandability.

It is the only low cost Motorola 68000 computer with both extensive input and output facilities, and full bus expansion, on the Australian market.

Although the Applix 1616 is not designed specifically for office and business use, it does have processing, graphics and communication capabilities that are the equivalent of modest office desk top computers.

- The 7.5 megahertz Motorola 68000 processor used in the Applix 1616 is identical to that in an Apple Macintosh Classic, but runs about 25% faster. The slightly more advanced 68010 processor can also be used, without any other alteration. A 15 megahertz version is available as a $120 upgrade, and any Applix 1616 can be modified to this speed.
- Computing speed is faster than an IBM PC XT, but not as quick as an IBM AT.
- The keyboard is a full IBM style keyboard, not a cut down version with no cursor keys. Almost any IBM XT style keyboard can be used.

- The bit mapped graphics provide the same definition as the IBM CGA standard (but with more colours on screen). The standard display is 320 by 200, in 16 colours, or 640 by 200 in a choice of 4 colours out of a palette of 16. These can also be displayed as 16 grey scale levels on a suitable monochrome monitor.
- IBM EGA standard graphics are also available, if you have a suitable display screen. This provides 640 by 350 displays, in 4 colours, or 320 by 350 displays in 16 colours (a simple modification to the 1616 is required for use with some brands of EGA monitors).
- 960 by 512 high resolution graphics are available after a $29.95 single chip modification. Includes free Bellcore Mgr windowing package and mouse support written by Stephen Uhler.
- Two standard RS232C serial ports are provided, using the same components as in the Macintosh. Two additional serial ports are also available on the disk drive controller. A Televideo 950 terminal emulation program is inbuilt. Full handshake control is a standard feature.
- A standard Centronics parallel printer port is provided. The text editor can support any brand of parallel printer.
- An Apple II compatible joystick port is included.
- Stereo sound outputs are standard, and include a two or four watt amplifier. Macintosh and Amiga sound files can be played without any additional equipment. Sound can be digitised via the User Port by adding a microphone.
- Analog and digital input and output ports are available as standard, to measure and control the outside world.
- The optional disk controller allows the use of both floppy and SCSI hard disks, and adds the ability to run ZCPR3 (an advanced variation on the CP/M operating system) programs. Using a utility program, it can read and write Microbee and IBM disks. All serious users should aim at getting one.
- The optional memory board allows up to four megabyte of memory, with optional ‘virtual memory’ support for Minix, plus an additional optional high speed SCSI disk controller port. For those with big program needs.
- The optional high resolution video board includes over a half megabyte of video memory, and a TI32010 graphics processor. For those into serious video work.
- Being prototyped, a combined Ethernet and memory board, providing four megabyte of memory, and Ethernet connections to mainframe or mini computer networks. Will include TCP/IP network software.
- Four expansion bus spaces, for add on cards.

Programmer or user?

Because the 1616 is not a standard computer, we assume that the type of person who buys it is either a programmer, or is trying to learn to program. Therefore **the internal working of all commands are always available** to you. In contrast, most computers assume you are not a programmer, and hide their internal workings from you.

Most office computers, such as the IBM and its clones, are designed with the assumption that they will be programmed only by professional computer programmers. Although they are not deliberately made hard to use or program, it is often difficult for a new programmer to find sufficient information to start doing their own programs. As a result, it is unusual for users to become involved in writing programs; most buy brand name application programs from the wide (and often expensive) range available.

Buying application programs works well, provided you can find programs that do exactly what you wish to do. However if your requirements are unusual, you will be forced to write your own programs. Many Applix 1616 owners bought their systems because they could not get their job done conveniently using ordinary office computers.

When writing programs, the Intel central processor chip (CPU) used in most business machines has many restrictions that even professional programmers often find troublesome. In contrast, the Motorola 68000 chip in the Applix 1616 is usually accepted by programmers as being flexible, and (relatively) easy to use. It has some style and elegance, lacking in many such chips. This makes Motorola based computers especially suitable for many educational needs.

Although other computers, like the Apple Macintosh, use the 68000, they are still not always easy to program. In attempting to make them simple for the user, the Macintosh forces a certain method of working on the programmer. The Macintosh and similar graphic interfaces generally require extensive knowledge and experience before satisfactory programs can be written. It is sometimes said (by programmers) that the Macintosh manuals 'contain numerous chapters, all of which are perfectly clear, provided you already understand the rest of the book!'

In contrast, the Applix 1616 is designed to make things easy for the new programmer. Like most micro computers, the Applix has a variety of inbuilt commands that can be typed, and which cause the computer to do certain common tasks. Unlike most other computers, these commands can usually be reached from **within** other programs, with great ease. Programs in the Applix are **designed** to work together. The result is highly similar (on a smaller scale) to the UNIX operating system in approach.

Each inbuilt command is made up of a number of smaller, less comprehensive or less flexible, operations. These operations are known as **system calls**, and many can be used almost as easily as commands. These **syscalls** are, like regular inbuilt commands, also available from within programming languages, and within other programs. Similar methods are used by all computers, however the system calls are often difficult to use, or are 'hidden' from the user.

The result is that, in the 1616, many programs can be partly written by 'pasting' together a variety of commands and syscalls, rather than writing everything yourself. Although similar methods may be available in various languages for other computers, each language is different in how it deals with the machine. In the Applix, once you learn how to use some inbuilt facility in one language, you can generally use it in any other language.

About the 1616/OS operating system

1616/OS was written by Andrew Morton of Applix. The code in the 1616's EPROMs is really a combination of a disk operating system, an input/output system and various small to medium sized application programs. A partial list of what it does includes:

- Hardware test routines, used during construction, and to locate faults.
- Facilities to call external ROMS, for use as a dedicated controller in industrial applications.
- Drivers for input and output devices, including printer port, two serial ports, cassette port, stereo sound outputs, D/A and A/D converters, etc.
- System timer and interrupt support, time and date drivers.
- Floppy disk, SCSI hard disk, and RAM disk support, including hierarchical directory structures.
- Bit-mapped video drivers for 320 and 640 column displays, windows, and clipping support for graphics, full cursor control via escape codes, plus display character set.
- A powerful line editor, with 'history' facilities, and function key re-definition for macros.
- WordStar-compatible full-screen editor with search and replace, block copy, move and delete, file merging, partial screen freeze, etc.
- A command line interpreter with over 60 commands, including machine code monitor, full I/O redirection, wildcard expansion, and access to over 200 internal system calls. It features many utilities, including an ASCII table, command help, and terminal facilities.
- Multi-tasking and multi-user facilities, to allow up to three users to simultaneously run up to 60 programs.

1616/OS is described as a disk operating system because one of its major functions is to manage the orderly storage and retrieval of programs and data on block storage devices such as floppy disks and hard disks.

Material relating to programming 1616/OS is found in the *Programmer's Manual* and the *Technical Reference Manual*.

A note from Andrew Morton

1616/OS is written almost entirely in the C programming language. It is a fairly large program (the source code occupies over 3/4 of a megabyte) and I grudgingly concede that there may even be a bug or two in it. If you think you have found a programming error, please contact me at Applix, preferably by sending sufficient material for the suspected error to be reproduced.

Enhancements, changes

I have attempted to make the ROM code as flexible as possible, allowing your programs to control anything which they could possibly want to without having to directly manipulate the 1616's hardware or the operating system's data areas. If you encounter limitations in the existing software or if you simply wish to see new features added, please write down your thoughts and send them along to me.

Standards

A problem which is seen time and again with computers of all sizes is that of data compatibility. Trying to transfer a file which was created under company XYZ's database program over to company ABC's word processor and then patching in a picture created under company BPL's paint package can be relatively painless, as long as standards exist for the representation of the data files.

Applix has not attempted to define a 'standard' representation for a word processor file, a picture file, a compressed file, a sound file, a spreadsheet file, etc. What we propose is that you contact Applix before embarking upon an implementation of one of these things. If we know of, or have defined, a suitable data structure we will provide you with a description of it; otherwise we will use your proposed format (or a generalisation of it) as the future standard.

Upgrades

At Applix, we wish to keep you up to date as new features are added to the 1616's operating system, particularly if these involve compatibility changes. However the cost of EPROMS, and printing extensive documentation, and the time involved in preparing them, means that a charge must often be made. A small refund may be made for old EPROMS, if their size is such that they can still be used for current upgrades. Contact Applix for current prices and upgrade details.

The upgrade from Version 3 to Version 4 is $29.95. These changes include updates for your manual sets, plus an updated User Disk. Within versions, changes are often made (Version 4.1a to 4.2d, for example), and updates within Versions are $5, **provided the original eproms are returned**. Manuals are not updated with $5 upgrades. Disks may be an additional $5. The latest version printed manuals are available at any time at $10 per manual. If you have printing facilities, the

ASCII text of manuals or a Postscript file of each manual can be made available on 1616 or IBM disk (please note that Postscript files in particular are very large, typically one manual per disk).

Conventions

To avoid repeating how to use certain key combinations, or types of commands, we have used various typographic conventions throughout this manual. These are detailed below.

Keyboard conventions

Throughout this manual, we assume that the reader can type, even if only slowly. We also assume some familiarity with typical computer keyboards, and that you can locate the

- **Control**,
- **Alt**,
- **Esc**,
- **cursor** (arrow) keys
- **function** (F1 to F10) keys.

We assume you will remember to turn the

- Caps Lock,
- Num Lock, and
- Scroll Lock

keys on and off as appropriate when using the numeric keypad for number entry or as a cursor pad.

We assume you will remember to let the computer know you have completed each command line by pressing the Enter or Return key.

We assume you know the **cursor** is the little blinking (or perhaps steady) box (or perhaps line) that can be moved about the video display.

In fact, we assume far too much, if you are really a total beginner. If you don't know about all the above, try to find someone (perhaps by attending any computer User Group or evening college) who can show you what the keys are, and how they work.

If all else fails, simply experiment. It is remarkably hard to damage a computer by typing something wrong. Merely learn where the **reset** button is (at the rear of the computer system box), and use it freely if and when you have problems. The Applix system survives a normal reset with very few problems. Your ram disk contents will remain intact, and your directory and other settings will be unchanged.

Extra keys

Although a computer keyboard typically contains between 80 and 110 keys, the computer itself can (and often does) use the equivalent of 256 different characters (2 to the 8th power ... the *why* of this particular number will eventually be obvious). We therefore may need means of typing all these extra characters.

Just as you use [Shift] keys to change ordinary alphabetical keys between UPPER and *lower* case, the [Ctrl] and [Alt] keys are used to produce characters that are not normally on the keyboard. You can also produce any of the 256 computer characters by using the numeric keypad.

Control key

A **control** character is often indicated in computer manuals by preceding it with an up-arrow character, eg **^A** is a control-A. This method is used in the Edit Quick Reference in Appendix A of the *User Reference Manual*, to save space. Normally however the **Control** key will be shown as [Ctrl]. A control character is entered by holding down the [Ctrl] key whilst typing and releasing the appropriate character, in this case an [A]. It does not matter whether the character is in upper or lower case. The line editor uses many control keys.

Alt key

Similarly, the symbol [Alt][A] (**alternate** A) indicates holding down the [Alt] key then pressing [A]. Several 1616 commands make use of the [Alt] key.

Esc key

The notation [Esc] indicates pressing the key labelled **Esc**, releasing it, and then pressing the next key required. The [Esc] key precedes use of another key. It is not often used like a shift key.

Number conventions

A '**$**' sign in front of a number indicates that the number is in base 16, or a hexadecimal number. However you **do not** type the $ sign when entering the number from the keyboard. Hexadecimal and binary numbers are explained in detail in many books on programming, and are traditionally used by programmers, mainly because programmers find them easier and more convenient than 'ordinary' numbers.

An ordinary decimal number is indicated by having a period '**.**' placed in front of it. A binary number is indicated by a '**%**' in front of it. Since it is more convenient (for programmers) to work in hexadecimal, all numbers entered into the 1616/OS are assumed to be hexadecimal, unless you indicate otherwise. If entering a decimal number from the keyboard, you do so by placing a '**.**' in front of it.

If entering a binary number, you place a '**%**' in front of it. This feature may confuse you at first, however persistence will pay off in learning these conventions.

Typeface conventions

In general in these manuals, commands that you type, the names of files, and example programs, are shown in `courier typeface`. The names of ***system calls***, or ***syscalls*** are generally in ***bold italic*** face. Internal variable names in system calls are generally in helvetica typeface.

File conventions

A 'pathname' is the name of a disk file or directory. A 'relative pathname' is the name of a directory, relative to the current one. An 'absolute pathname' or 'full pathname' is the full specification of a file or directory's place in the file system. The concept of 'pathnames' is very important, and will be explained in the section on Disks, Files and Directories.

Filenames and any arguments that are optional are usually enclosed in square brackets thus `[ -v]`. This indicates something that you can type, to cause a certain action, but you are not forced to do so for the command to work. If more than one filename can be used in a command, the repeat action is indicated by `...`

2
First Steps

In this chapter, we assume that you have a completely built and tested 1616 (preferably complete with 3.5 inch disk drive) ready to be unpacked and plugged in. Kit builders will already be familiar with connecting the 1616, from their *Hardware and Construction Manual*.

This chapter explains how to start using the Applix 1616. It lists the connectors available for external devices (peripherals) such as printers, modems, joysticks, and so on. It explains how to use the keyboard, and the numerous extra features of the Applix keyboard drivers, such as recalling past commands, and function key macros.

The Applix 1616 includes a powerful line editor, that makes it easy to alter very long commands, and this is also explained. Please learn how to use the keyboard; the Applix keyboard handling is specifically designed to make life easier for those of us who are not fast (or accurate) typists!

Starting the 1616

Starting the Applix 1616 is easy. Connect everything, and turn on the power. The operating system (and almost everything explained in this manual) can work perfectly well without a disk drive. However we do suggest that the computer is much more pleasant to use when fitted with a disk drive.

Connecting everything

At the rear of the 1616 are a large number of connectors, and switches. Looking at the rear, and viewing from left to right, these are:

- Keyboard (5 pin DIN socket)
- Loudspeakers (5 pin DIN socket)
- Cassette (5 pin DIN socket)
- Reset button
- Joystick Port (9 pin D socket)
- Serial Port B (9 pin D plug)
- Serial Port A (9 pin D plug)
- Video Connector (9 pin D socket)
- User Control Port (34 way plug)
- Centronics Parallel Printer Port (26 way plug)
- Power Switch
- Power Line Connector (3 pins in rectangular socket)

Plug an IBM XT compatible keyboard into the left hand 5 pin DIN socket. If there is a switch on the underside of the keyboard, it must be set to the ‘XT’ position, not the ‘AT’ position, before switching on the power to the computer.

If you have other optional peripherals, such as mouse, printer, loudspeakers, plug them in to the appropriate connectors.

Plug the lead from your monitor (Apple style monochrome, or IBM RGBI CGA colour, or multisync) into the rightmost 9 pin D socket, and switch on the power to the monitor, and any other peripherals such as printers, but not the computer.

Switch the power on to the computer.

Starting the system

The Applix 1616 will start without a disk in a drive, but it takes longer, and limits what you can do to only those commands described in this manual. You will be limited to using the inbuilt ram drive /RD to store files, and the contents of the ram drive are destroyed whenever you turn off the power.

If available, you should use a 1616/OS version 4 boot disk (you received a suitable disk labelled 'User Disk' with the 1616 disk co-processor card) in any disk drive (it is traditional to use /F0 - this is the one whose light comes on first when you switch on the power - as many non-Applix computer systems are unable to boot from any drive except their first drive).

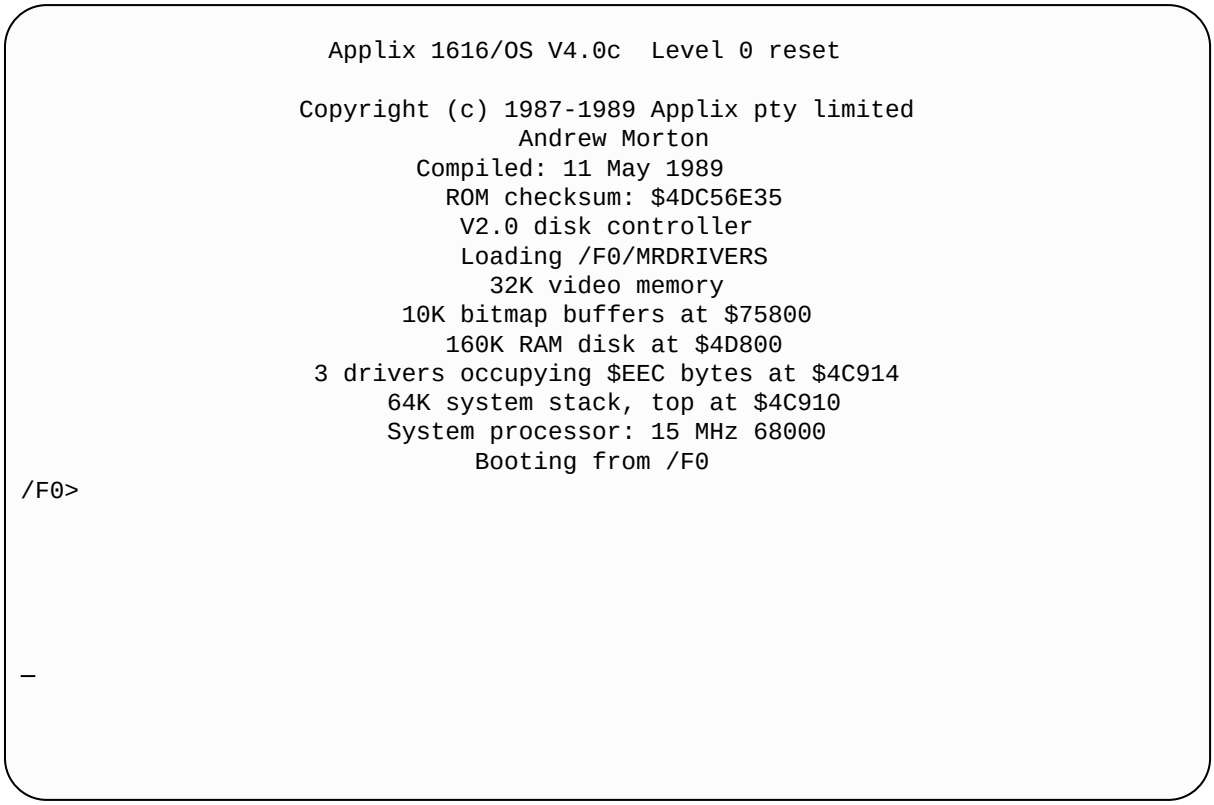
Those with disks from older versions of the operating system should note that **version 2 disks will not work** as boot disks. Using a non-boot disk will stop you from starting the Applix; if you are not sure you have a proper boot disk, just start up without a disk in a drive for your first time.

Insert the disk in the first disk drive, with the shutter on the disk at the front. As you push it into the drive, it will click into place. If the disk will not fit correctly and does not click into place, you are probably putting it in upside down. Never force a disk into a drive.

Switch on the power for the display, and then for the computer (the switch is on the rear, towards the left hand side).

The disk drive light will flash for a while. You should see a number of messages on the display, including one about 'booting from /F0'. Finally, you will have a prompt on the display, showing either **`/RD`** or **`/F0`**. Your prompt may also show that you are in a particular sub-directory, such as **`/F0/BIN/`**. After the prompt you should have a flashing cursor.

If you do not have such a prompt, and are using disks, re-read the more detailed note that accompanies 1616/OS Version 4, on starting the system. In newly updated systems, the most likely problems are not using a Version 3 or 4 boot disk, having the jumpers for your eproms wrong, or having installed new eproms incorrectly.



The keyboard

This section describes how to use the 1616 keyboard at the command line level. This is when you are staring at a screen display that contains a prompt starting with / such as `/F0` or `/RD`.

The keyboard is explained first, as it is the primary input device for the 1616. The keyboard and line editor handling of the 1616 are very powerful, especially by comparison with conventional business systems such as MS-DOS, and it is worthwhile learning all their features. In particular:

- the most recent lines that have been entered can be recalled for editing as in some versions of UNIX,
- multiple commands can be typed on a single line,
- commands can run in background, while you continue typing your next command,
- the function keys can be readily altered to print up to 63 keystrokes, without any additional program being required.

Using the 1616 keyboard

The 1616 keyboard is interrupt driven, which means that 1616/OS can receive keystrokes immediately, and save them to be presented when they are needed. This means that you may type when the 1616's microprocessor is not actually

awaiting keyboard input; the keystrokes are not displayed upon the screen until they are used. Up to 200 keystrokes may be accumulated, which is far more than most systems allow. Experienced users can alter this value to virtually any desired level using the ***new_cbuf*** system call.

Keyboard failure

The Applix 1616 is designed to work with any IBM XT compatible keyboard with a 5 pin DIN connector. Applix can supply a suitable keyboard, but in these days of low prices, many users prefer to obtain their own. Unfortunately, such keyboards do differ somewhat from brand to brand, so sometimes a new user encounters a keyboard problem.

If your keyboard does not work (does not type anything, or types the wrong thing), make the following checks.

Check that the small switch (if available) underneath the keyboard is set to XT mode. The Applix does not use IBM AT or IBM PS/2 style keyboards. Power down the entire system for about 20 seconds after moving the switch, to allow the keyboard to reset correctly.

There is a small set of two pins labelled KB at the front left hand side of the Applix motherboard. These can be used to adjust the Applix to some of the known different styles of XT keyboard. Switch off the Applix and remove the cover. If a small plastic jumper block is in place shorting out the two pins, remove it! If there is no jumper block, install one (or short out the two KB pins with wire). Try the keyboard again.

Try a different keyboard, if you have one readily available, or can easily borrow one. This tests that the Applix 1616 keyboard circuit is working.

If you are still having keyboard problems, contact Applix direct for additional help. Note that you will not be able to use an IBM AT only keyboard, you do need an older style IBM XT keyboard. A keyboard with an "automatic" AT/XT switch is also not likely to work (they are designed for specific IBM clone systems). Applix can always supply a keyboard that is guaranteed to work, if you can not easily locate one yourself.

Cursor keys and numeric keypad

The keypad on the far right-hand side of your keyboard has two modes. One is as a cursor control pad (**Num Lock** light off) and the other is as a numeric keypad (**Num Lock** light on). You may switch between the two modes by pressing the Num Lock key.

When the keyboard's numeric pad is in its cursor control mode, the 1616/OS keyboard driver produces the appropriate control codes for the line and screen editors. In short, you can move the cursor about the display.

The state of the Num Lock key is temporarily reversed when a Shift key is pressed: if the numeric keypad is in cursor control mode then holding down a Shift key will place it in numeric mode, and vice versa.

Control and Alt keys

Typing a key while the Ctrl key is depressed will generate the ASCII code of that key, minus 64, thus providing a Control code between 0 and 31 (for programmers, bits 5, 6 and 7 are zero). Many Ctrl key combinations are used by the line editor, and the full screen editor. A control key combination, such as Ctrl C, is often shown as **^C**, in manuals and books that can not show the actual key. We show it that way in many manuals for reasons of space.

You don't really need to know, for the purposes of this chapter, what each key produces. For example, Ctrl^ (the shifted 6) generates ASCII code 30, while Ctrl@ (the shifted 2) generates ASCII code 0. Later, when you learn programming, you will want to know these sorts of key equivalents (the Applix will produce a list of equivalents for you when you type the command `ascii`).

Typing a key while the Alt key is depressed will generate the ASCII code of the key, plus 128 (for programmers, bit 7 is set by the Alt key). Special exceptions to this rule of thumb are listed below.

All 256 characters of the extended ASCII character set can be produced by holding down the Alt key and typing the character's ASCII code (including leading zeros) on the numeric keypad section of the keyboard. This does not depend upon the state of the Num Lock key.

The ASCII character set is explained in detail in many elementary programming texts. It is simply a method by which the computer keeps track of the meanings of characters. You can see an ASCII table on the 1616 by typing the inbuilt command `ascii` (upper or lower case is fine, and don't forget to press Enter after the command).

The Alt key

A few special functions are implemented using the keyboard's Alt key:

Alt T — Typing this sequence at any time switches the cassette relay to the opposite state. Do not type this during cassette I/O! When the cassette relay is connected to a cassette recorder remote control, it switches the motor on and off.

Alt Ctrl R — This sequence is typed by holding down the Alt key, then holding down the Ctrl key, then the R key. The effect is the same as pressing the 1616's **reset** switch. This key sequence will have no effect if the 1616 has seriously hung up for some reason - you will then have to use the reset switch. This is used if the 1616 fails to respond due to a program failure, and you can not regain control in any other manner.

Alt S	The Alt S command stops and starts video output only. This may be used to slow output to a readable rate, assuming your reflexes are excellent. The flashing video cursor changes to an underline to indicate that output is suspended.
Alt Ctrl C	This is a powerful way to abort or escape from a program in foreground. It will escape from or abort the vast majority of programs. Use Alt Ctrl R to reset, if this fails.
Alt Del	Toggles the end of file (EOF) character. Default is $100 (that is, no end of file character), and the alternate value is $04 (or Ctrl D). See the `option` command to set EOF to any desired value. This EOF character is often used to indicate the end of input from devices such as the keyboard, or from other computers.

Function keys

The 1616 keyboard's ten function keys F1 to F10 may be used to produce up to 63 characters of input from a single keystroke. The `fkey` command may be used to define a function key from within a `.shell` program, as explained in a later section.

There is a manual capture mode on function keys: You hold down the Alt and Ctrl and press the function key. From this point all characters typed are invisibly captured into the definition for that function key. The definition may be terminated by pressing the function key which is being defined.

This feature is handy for 'remembering' a sequence of editor commands which needs to be repeatedly typed. If you find yourself repeatedly typing any lengthy group of commands, capture them on a function key, and use that instead. The captured function keys are lost when you switch off the system, and you would have to retype them. The `fkey` command is provided so you can save function key definitions on disk.

Last line recall and completion

The up-arrow ↑ and down-arrow ↓ keys (or Ctrl E and Ctrl X) scroll up and down singly through the last ten lines which have been entered in the line editor. Once you have found the line you wish, you can change it with the line editor.

As an additional feature, you can type the first few letters of a previous command, and then press the Esc key. The most recent command starting with those letters will be displayed ready for editing. Pressing the Esc key again will bring up the next most recent command starting with the letters you typed. This applies by default to the last ten commands issued. You can change the number of command lines retained by altering your `mrdrivers` file, as explained in the documents for the `mrdrivers` (*Technical Reference Manual* or *Users Disk Manual*).

In addition, you can type the first few characters of a filename, and then press the Tab key, and the 1616/OS will attempt to complete the filename you have started. You can also use Ctrl C, and the 1616/OS will attempt to match the last word. Likewise Ctrl O will recall the last word from the previous line, and if used again, the word before that, and so on.

The line editor

Whenever the operating system, or any program which runs under it, must obtain a line of input from the keyboard, the system line editor is used. The line editor permits you to enter a single line of up to 511 characters. It has many features to simplify and accelerate command entry.

In particular, the line editor permits you to recall to the screen any of the last ten commands you have issued (you can alter the number of commands it stores, if you have a disk drive). It then allows you to modify these commands, to make a new command. This feature allows very rapid changes to repetitive commands, and easy correction of mistakes in commands. You should learn how to use the line editor, as you discard much of the power of the keyboard if you ignore it.

The line editor's commands have been designed for compatibility with those of the screen editor - only the Ctrl Y command differs. The line editor's special features are invoked by typing control characters. These commands generally follow the same pattern as the 'WordStar' word processor.

To insert characters into the line one types in the normal manner. The Ctrl W key causes previously entered lines to be inserted into the current line at the cursor position. Typing it once causes the most recent line to be inserted. Typing it again causes the addition of the line before that and so on.

The line editor's control character commands are grouped as follows:

Cursor positioning commands:

Ctrl D or →	Go forward one character
Ctrl F	Go forward one word
Ctrl S or ←	Go backward one character
Ctrl A	Go backward one word
Ctrl B or End	If at start of line, go to end; otherwise go to start

Text deletion commands:

Ctrl G, Del	Delete the character under the cursor
Ctrl T	Delete from the cursor to the start of the next word
Ctrl Y	Delete from the cursor to the end of the line
Bs, Ctrl H	Delete the character before the cursor
Ctrl V	Delete from the start of the line up to the cursor

Miscellaneous commands:

Ctrl W	Recall previously typed lines, insert most recent at cursor

Ctrl P	Escape a control character - after typing a Ctrl P you may enter any single control character into the line. Use in Function Key definitions also.
Ctrl M, Enter	The Enter key is pressed when you are satisfied with the line. The cursor may be at any point on the line when Enter is pressed. On some keyboards, the Enter key is marked Return.
Ctrl C	Match last word.
Ctrl O	Last word of previous line, then word before, etc.
Tab	Complete a filename, if possible.
Ctrl E or ↑	Scroll backwards through previous lines
Ctrl X or ↓	Scroll forwards through previous lines.

The Ctrl A, F and T commands work on a word by word basis. A word is considered to end with a space, tab, . ! , : ; ( ) [ ] { } / -

3
Command Handling

The 1616 is made to do things by typing commands at the keyboard. If the command is not correctly typed, then an error message is produced. A command is simply the name of a program, or the name of an inbuilt command or system call.

In many cases, the command or program requires *parameters*, or *arguments*. A parameter or argument modifies the effects of a command, by telling it which of several possible things you want done. Some commands accept more than one type of parameter. Some accept multiple instances of a single type of parameter.

This chapter explains the general form of commands, their syntax, and gives examples of some simple commands for you to try. The Applix can accept numbers in decimal (the way humans write numbers), hexadecimal (the default, as computer numbers are usually expressed this way) or binary (the way computers use numbers internally). This chapter explains how to use each form.

Punctuation and special characters are important when issuing computer commands; these are listed. The general way commands are executed (obeyed) by the computer is covered. A brief explanation is given of the very powerful redirection facilities of the Applix 1616. Wildcard expansion is explained, showing how you can avoid typing full details of filenames, and instead let the computer work them out.

General command format

The general format for the entry of commands is:

```
commandname  arguments  <inputsource >  outputdestination  }
    errordestination
```

'Commandname' is the name of the command (see the full list later in this manual) which is being used. We show names of commands and names of files in a `courier` typeface, to make it easier to identify them in the manual.

'Arguments' refers to any additional instructions or data which the specific command needs.

The special redirection operators <, >, and **}** are explained later. They make it possible to use computer files, or devices other than the keyboard, for entering material. They also allow the results of the command to be displayed somewhere other than the normal display screen, such as on the printer, or on another computer via a serial port.

Command input syntax

1616/OS scans the line that you type, responding to various characters as appropriate. For example, if you use one of the line editor control codes (see Section 2), then the line is edited. However, when you press the Enter key, the operating system then tries to carry out the commands entered.

Spaces and tabs separate the commands and all the arguments from each other. This means that you can not leave spaces in filenames, or it will appear that there is more than one word. 1616/OS will assume that the first word in the line is a command, and attempt to execute that command, as detailed later. For example, type the inbuilt command

```
ascii
```

and you will get an ASCII table on the display.

Examples of commands

Type the inbuilt command

```
date
```

and the date and time (probably incorrect, see the inbuilt `setdate` to fix them) will be displayed. To obtain a list of available inbuilt commands, type

```
help
```

If a command takes arguments, these come after it, separated by spaces. Try

```
help setdate
```

The `help` command, with `setdate` as an argument, now tells you that the inbuilt `setdate` command requires a series of values as its arguments. Change the date and time by using

```
setdate .92 .12 .21 .14 .30 9
```

but substitute the correct year, month, day, hour, minute and second. Note that the year takes only two digits, and that the hour is in 24 hour clock notation (.14 is 2 p.m.) Notice also that where a number exceeds 9, I placed a **.** in front, to indicate I intended a decimal number (even computer people tend not to express dates in hexadecimal!)

Each command ‘knows’ what sort of arguments it needs, and will give an error message if you provide the wrong type of argument. Now type

```
time ascii
```

The inbuilt command `time` simply measures how long something takes to happen (experienced users may note this is the way the equivalent UNIX command works - MS-DOS gives you the less useful time of day instead). As with the `help` command, its arguments happened to be yet another command.

Typing in numbers

Whenever a number is to be entered on the command line as a parameter to an inbuilt command there are three ways of representing it:

- Decimal numbers are entered by preceding them by a full stop '**.**'.
- Binary numbers are entered by preceding them with a '**%**' symbol.
- Hexadecimal numbers are entered without any leading character.

Decimal numbers may be preceded by a minus sign, eg. .-1234

As an example, the command

```
mfb 4000 .17000 %101011
```

will fill the memory range 4000 hexadecimal (16384 decimal) through 17000 decimal with the binary value 101011 (decimal 43) (assuming you have some reason to want to do this!)

Numeric commands

Most commands have as their arguments filenames, or numbers. Try the inbuilt expression calculator (a poor man's pocket calculator)

```
expr .23 + .84 - .29
```

If you don't agree with the answer, note that you **must** put a '**.**' and mathematical operators in front of decimal numbers when entering them, and must leave spaces between all arguments. The answer is given in binary, decimal and hexadecimal. Check the values of different base number with the inbuilt `base`.

```
base 40 .60 %100100
```

This command converts a set of numbers into hexadecimal, decimal (indicated by the **.** in front), binary (shown with **%** in front) and in addition, to octal. If you don't know about different base number systems, you can read up on them in many computing and math text books.

You should also note that these commands only work with integer numbers. That is, you can't use numbers that include a fractional part, or a decimal point.

Special characters

The semicolon character '**;**' is used for putting comments on command lines (particularly within `shell` files, see below); any characters occurring after a semicolon are ignored. If the first non-blank character in a line is a semicolon, then the entire line is ignored. Use this method to remind you what a line is supposed to do.

If you actually wish a semicolon to appear on the command line, rather than using it to indicate a comment, it will have to be surrounded by double quote (") characters. For example, note the difference between the two examples of the inbuilt `echo` command, which merely repeats its arguments

`echo abc;def` and `echo "abc;def"`

If you wish to enter a quote (") character at the command line, type in two successive quotes.

Wherever possible, the operating system does not distinguish between upper and lower case characters on the command line. The words on the input line may be separated by any number of spaces or tabs. Spaces or tabs before or after the command are ignored, but at least one space must always separate a command from its arguments.

A '!' symbol may be used within a command line to separate multiple commands, so you can write a lengthy command sequence easily.

```
echo abc ! dir ! base 40
```

On some occasions, such as when defining a function key with the `fkey` command, you will wish the system to consider a group of characters which includes spaces and tabs to be a single argument. This is done by quoting the group of characters. For example the command

```
fkey 2 "spaces->  <- here"
```

preserves the spaces in the string of characters, for use in the definition of function key 2. This technique allows you to define function keys from within shell files, rather than doing it from the keyboard with the Alt Ctrl function key method.

Error messages produced by 1616/OS

1616/OS produces error messages which should be self-explanatory in the context of the command which was being executed. Refer to the documentation for the particular command if the error message is not sufficient.

Occasionally the system may emit obscure looking error messages from a lower level. These error messages, which result from errors and inconsistencies in user programs, file systems or possibly 1616/OS itself, are described in an appendix to the 1616/OS *Programmer's Manual*.

Command execution

There are three basic types of command available:

- executable memory resident drivers
- inbuilt commands
- transient commands.

When a command is typed in and the Return or Enter key is pressed, 1616/OS compares the command name with the names of all the currently installed memory resident drivers (MRD) which can be executed from the command line. If a match is made, the MRD is called to handle your command. This sort of command is referred to as an 'executable memory resident driver'. These commands *modify* the way the system works. They are installed from disk by the `mrdrivers` file when you first start the system each day (if you have a disk drive, you may recall seeing messages about them on your display when you started the system). These

MRDs provide a powerful method of modifying every detail of how the computer works. You can even replace existing commands with MRDs, or expand commands to make the system more powerful. Use of MRDs are explained in the *User Disk Manual*, and we try to explain how to write them in the *Technical Reference Manual*. Some common MRDs are the ones that blank the display when it is left unattended (`crtsav`), the one that displays the time (`tdos`), and the one that provides an EGA and extended graphics display (`sseg`).

If the command did not refer to an MRD, the command name is compared with the 60 or so which are implemented within the 1616/OS program. These are the commands listed when you use the `help` command. If a match is found, the selected function is executed. The actual MC68000 microprocessor code to perform the selected command is contained within the 1616/OS EPROM chips. This sort of command is referred to as an 'inbuilt command'. It is these commands that we describe in this manual.

If no match is made with the inbuilt commands then the operating system searches for disk files with names based upon the command which you typed in. If, for example, you typed in '`diskcopy`', the following searches occur:

'`diskcopy.xrel`' in your current directory (the directory that appears in your prompt).
'`diskcopy.shell`' in your current directory.
'`diskcopy.exec`' in your current directory.

If none of these are found, then a search is made for these files on other directories, as specified by your current execution path (see the `xpath` command for details). Most users make up a special directory, usually called **bin**, to contain programs they often use. Upon starting the system, they use something like the command `xpath + /f0/bin` to ensure that the directory **/f0/bin** is always searched, as well as their actual current directory.

If one of these files is found, it is executed. `xrel` and `exec` files contain executable MC68000 machine code. This sort of command is referred to as a 'transient command', because the microprocessor code for it is loaded into memory from a disk device only when it is needed. The `shell` file contains a list of commands (such as you would type in from the keyboard), which the system interprets. Many of these disk based commands are discussed in detail in the *User Disk Manual*.

Input / output redirection

The optional input / output redirections '<', '>' and '**}**' in the command line format described at the start of this section will get input from, and send output and error messages to, the nominated character devices or files for the duration of the command. Using the doubled redirections >> and **}}** means that instead of overwriting the previous output, new output is added to it.

When you redirect I/O for a command, you effectively replace the keyboard and the video display with different character sources and/or destinations. When the inbuilt or transient command is executed it will typically send its printed output to 'standard output', which is normally the video display. Input characters come from 'standard input', which is normally the keyboard. Error messages should go to 'standard error' which is normally the video display. I/O redirections reassign standard input, standard output and standard error to different physical devices or to files. The advantage of this sort of thing is that you can save the output of programs in files, or on a printer, or even send it to a different computer. Likewise, you can use a file to provide the input to a command or program, instead of typing something repeatedly from a keyboard. Great stuff for lazy people!

This works for both inbuilt and transient commands, and is a very powerful tool. The '<', '>', '>>', '**}**' and '**}}**' constructs must be the last part of the command line; all characters after these are ignored. Do not place both the '>' and '>>' constructs, or both the '**}**' and '**}}**' constructs on the same line, as this will not produce sensible results. The redirections may be entered in any order. Using wildcards in I/O redirection filenames will not work; the whole filename must by entered.

A character device is identified by a name followed by a colon. No differentiation is made between upper and lower case in the device name. The character devices available when you turn on your 1616 are:

CON: The CON: device, or console, is the video display when used as output and the keyboard when used as input.

SA: The SA: device is serial channel A for input and output.

SB: The SB: device is serial channel B for input and output.

CENT: The CENT: device is the Centronics parallel printer output port. Input may not be obtained from this device.

NULL: The NULL: device discards characters which are sent to it. Directing the output or error messages from a program onto this device will prevent them from being displayed. This can be handy for throwing away unwanted error messages, or `syscall` results. Input may not be obtained from this device.

TTY: The TTY: pseudo device is intended for use by programs expecting interactive input from the user, where input redirection may be involved. The pseudo device makes an inspired guess at the required input device, and acts as if that is standard input. This is handy for running some time consuming program in background mode, while still feeding it commands (formatting disks using `blockdev` while doing something else, for example).

Some examples of commands which employ I/O redirection follow:

```
dir > myfile
```

The `dir` command displays the directory on standard output (normally the video display). When 1616/OS receives this particular command it creates a disk file called `myfile` and directs all the output from the `dir` command into the file. If the file already exists, this will overwrite (and thus destroy) the old contents of the file.

```
dir >>myfile
```

This command differs from the previously described one in that the resultant directory is appended to the end of the file `myfile`, rather than being put into a fresh file.

```
cat con: > filename
Any text you like
Ctrl D
```

This copies whatever is typed on the keyboard into a file, stopping only when you type the End Of File (EOF) character. The EOF character will not normally be set, but can be changed to Ctrl D, by using `option 6 4`.

```
edit myfile <edcommandfile
```

This command uses the full-screen editor upon the file `myfile`. The file `edcommandfile` would contain a sequence of characters which are presented to the editor as if you had typed them in.

```
SSASM asmfile.s -l >CENT: }errorlog
```

SSASM is the M68000 assembler for the 1616. In this example, it assembles the file of assembler code `asmfile.s`. The assembly output is sent to the parallel (centronics) printer port. Any error messages are recorded in the file `errorlog` in the current directory.

Another redirection facility available is piping. This is used to link many commands together. The result of the first command becomes the input of the next command, and so on. It is these redirection facilities that enable you to 'build' very powerful and complex commands out of a series of simple commands.

For example, you might use some UNIX style disk utilities to generate a phone list for a visit to Victoria from a more extensive address list, with a line similar to:

```
cat address.list | grep Vic | cut -f1-2,7 > phone.list
```

which would throw away any line not containing Vic, then take the first, second and seventh space delimited fields of what remains, and put the result in another file.

Wildcard expansion

Often you will wish to use an inbuilt or a transient command to perform some operation upon a group of files. There will frequently be some similarity between the names of the files. To specify such a group of filenames you may enter a

prototype name which contains a mix of characters and wildcards. The wildcard '*****'matches any group of zero or more characters; the wildcard '**?**' matches any single character. What all this means is that you can specify names without typing them in full, and manipulate groups of similarly named files without specifying each of them individually. Basically, it saves you effort and typing.

Wildcards which appear somewhere within double quotes are not expanded. If no match is found with your prototype filename it is passed unchanged.

Wildcards are only expanded in the last part of a pathname, so the specification `/F0/*/fred.s` is incorrect. The specification `/F0/srce/*.s` is correct because only the last part of the pathname has a wildcard.

Some examples:

- To delete all of the editor backup files from a disk you would type `delete *.bak`. The system reads the names of all the files on the disk and attempts to match each name with the prototype. If a match is made the full filename is substituted into your command line.
- To delete the files 'file1' and 'file2', but not 'file34' you would type `delete file?`. The question mark matches the single characters '1' and 2' but not the double character '34'.

To experiment with wildcard matching you can create files by typing `>filename` and then use the inbuilt command `echo` to find matches. For example, `echo *` prints out all filenames; `echo *c?` prints out all filenames which have a '`c`' as their second last character.

Wildcard expansion stops when the template is matched, so you can have a wildcard early in a filename, but restrict matching to a series of letters later in the filename. For example `*book.doc` will match only files that end in 'book.doc', and ignore files that don't have 'book.doc' in them.

Since wildcard expansion is done by 1616/OS, rather than by individual commands, it is entirely consistent from command to command. That is, wildcard expansion by `dir` is the same as by `echo`, and is the same as by `delete`, `type`, `cat`, `move`, `copy`, and so on. Some operating systems, such as MS-DOS, get this wrong, and therefore don't do wildcard expansion correctly (or at all, for some commands such as `type`).

In addition to the simple wildcards `*` and `?`, the Applix 1616 allows "grouping" of letters by means of the brackets `[` and `]`, and these can be used to indicate a range of letters for matching. For example `echo [a-c]*` will match any file names commencing with the letters a to c. The same effect occurs using `echo [abc]*`. You can also negate the effects of this (and thus exclude a particular range) by using ~ before a group. Thus `echo [~a-c]*` will show all file names that **don't** commence with a to c. Similarly, `echo [a-d][x-z]*` will match any filenames starting with a to d, followed by x to z as a second letter. The best way to learn how this works is to experiment with the `echo` command, and a group of junk files.

Command line processing order

To understand how command lines are processed it is necessary to know the order in which operations occur. When you have typed in a line and pressed the return key the following things happen:

1. Multiple commands, separated by '!' characters are separated for individual processing.
2. Any comments (indicated by semicolons) are removed.
3. The '<', '>', '>>', '}' and '}}' commands are searched for and interpreted. Any necesary file creation or opening is done and the redirection part of the command line is stripped off.
4. All wildcards are expanded and inserted into the command line in place of any prototypes.
5. The system attempts to match the command name (the first word on the command line) with one of the current memory resident drivers. If a match is found the MRD is called. Otherwise a search is made of the inbuilt commands. If a match is found the syntax of any arguments is checked with that which the particular command requires. If no error is detected the command is executed.
6. If the command is not an MRD or an inbuilt one, the system assumes that you are executing a file whose name is the same as the specified command, with `.xrel`, `.shell`, or `.exec` added. A search is made in the current directory, and in all the directories specified in your execution path (see the `xpath` command), for one of these three files.

4
Files and Directories

This chapter explains about block devices in 1616/OS. Block devices, despite their crass name, are one of the really handy things about computers, because they help you manipulate large chunks of information whatever way you need. Block devices store the contents of the computer memory in chunks called blocks, each of which is 1024 bytes long. They include disk drives, hard disks, and ram disks. In contrast, character devices such as the keyboard and printer manipulate memory contents on a byte at a time basis.

This chapter explains that computers store information in named files, and groups the files in directories. It explains the different types of block devices, such as ram disks, disk drives, and hard (or fixed) disks, their advantages and disadvantages. It tells you about hierarchical file systems, directory names, filenames, and pathnames. It introduces some easy commands for using block devices.

Although you do not actually have access to a disk drive or hard disk until you expand your Applix 1616, the inbuilt ram disk (known as **/rd**) obeys exactly the same commands, so it is possible to run the system without a real disk drive. However, the contents of **/rd** are destroyed once you switch off the computer, making this an insecure system. Although cassettes were the only choice a decade ago, today all serious users try to obtain disk drives as soon as they can.

You will find more detailed discussion of setting up disk drives in the *Disk CoProcessor Manual* and the *User Disk Manual*.

Background

Since the contents of the memory in a modern computer disappear when power is switched off, some method of storing files and programs more permanently is required. This method is, unfortunately, almost always on some magnetic media (and thus relatively slow by comparison with the electronics in the computer). The magnetic media the 1616 can handle includes cassette tapes, floppy disks, Small Computer Systems Interface (SCSI) hard disks, and other SCSI devices. SCSI is one of several standard styles of interfaces (connections) to magnetic storage devices for computers. Others you may encounter (not used by Applix) include ST506, SMD, ESDI and IDE.

Each file or program that is stored needs to have a name by which it can be identified by the user, and this is known as its **filename**. On the Applix, the filename can be up to 31 letters long, so you should use descriptive names. Upper and lower case letters are treated as if they were the same, while spaces and control characters are removed. You are not allowed to use the '/' character within a filename.

File types

You can have many different types of files, however for the moment, you can consider them to fall into three general groups. You can have programs, text or data files, and shell files.

If a file doesn't contain a program, it can be considered as either text or data. Sometimes it will be simply text, for example if a letter were produced using the `edit` command. Sometimes it may contain data that isn't particularly readable to humans, as in a file containing music, sounds, or pictures.

If you type the name of such a text or data file, you will get an error message 'Cannot execute filename: No such file or directory.' If you believe a file contains readable text, you can see the contents using the inbuilt `type filename` command. If the file does not contain ordinary text, `type` will still try to display it, however you may well get a real mess of display characters on your screen. Incidently, you should be cautious about trying to print such non-text files on a printer, as you can waste a lot of paper and upset your printer enough to have to power it down before you can use it again.

One special form of text file is a `shell` program file. This simply contains lines of normal commands, just as they would be typed from the keyboard. It has the filename extension '`.shell`'. When its name is typed, all the commands contained within it are executed or run, one by one, just as if you had typed them out at the keyboard. This can be a very convenient way of storing extensive commands that you tend to repeat often. As with '`.exec`' and '`.xrel`' programs, you must not type out the '`.shell`' extension.

Filename extensions

Programs always have names ending in an 'extension'. This extension consists of a full stop '**.**' followed by a term such as '`.xrel`' or '`.exec`'. You cause programs to run by simply typing their name. File names can be either upper or lower case. You must leave off the '`.xrel`' or '`.exec`' extension when typing their name. This rule of leaving off the extension follows traditional practice in computer systems, so you may as well get used to it (it does save some typing).

For convenience in identifying files, programmers often use names in which a '**.**' is placed in the name, and some agreed upon 'extension' is used to end the name.

Some extensions you will encounter, and must use correctly, are '`.exec`' and '`.xrel'`, which are programs. Shell programs, which you will learn about in this manual, have '`.shell`' as their extension.

Other common (but not mandatory) extensions are `.bas` for Basic language source files, `.c` for C language source files, `.doc` for program documentation, `.f` for Forth language source files, `.h` for header files, `.mac` for macro definitions, `.mus` for music files, `.s` for assembler source files, `.snd` for sound files, `.d`, `.text`, or `.doc` for various types of text files.

Block devices

As mentioned, block devices store the contents of memory in blocks of 1024 bytes. They include ram disk, disk drives, and hard disks.

The RAM disk

The basic 1616 system does not have any floppy disk drives. Instead, a proportion of the 1616's RAM memory is used as a disk; data is written to memory as if it were being recorded onto a disk. The simulation of a floppy disk using memory is referred to as a 'RAM disk'. The name of the ram disk is **/RD**.

There are many references to disks and disk files in this manual; the term 'disk' in this context does not actually refer to a physical floppy disk but rather to **any** mass storage device for which a software driver exists in the 1616. The RAM disk is one such device. Others devices can include one or two 3.5 inch (9 cm) or 5.25 inch (14 cm) disk drives, SCSI hard disks, or SCSI cartridge tape drives.

To ensure file integrity the RAM disk data is checksummed under software as it is written in. The checksum is verified when the data is read back; if some software (or hardware) malfunction has caused the RAM disk contents to be corrupted, then attempting to read the file whose contents were affected will result in an I/O error being reported.

You may vary the amount of memory which 1616/OS allocates for the RAM disk using switches 0 and 1 of the quad switch on the 1616's PCB as follows:

RAM disk selection	Switch 1 setting	Switch 0 setting	RAM disk size
0	off	off	24K
1	off	on	104K
2	on	off	200K
3	on	on	304K

It is suggested that you select RAM disk size 2 for the time being. If you have disk drives, then the settings in the MRDRIVERS file on your boot disk will override the switch settings as required. The ram disk can occupy up to a megabyte of memory, if you have this much to spare.

The major disadvantage of the **/RD** device is that the contents are lost when the computer is switched off. Why then use it at all? The answer is speed. A ram disk is faster than most physical disks. A typical use is to copy your most often used programs into **/RD**, and run them from there. Because you have a permanent copy of your programs on disk or tape, it does not matter that the copy in the ram disk is lost when you switch off. However, when you create a new file, or update the contents of a file, you want it saved in some less transient form. This will be on floppy disk or on hard disk.

Disk devices

All the disk devices on the Applix 1616 run identically. That is, the same commands apply. However, there are some differences in their speed, and capacity. The ram disk **/rd** is fastest, however it has the least capacity (typically 200k bytes, or about 40 pages of text), and the great disadvantage of totally losing its contents when the power is switched off.

A 9cm floppy disk drive (which accepts plastic encased diskettes) is slowest, but has a capacity of 800k bytes (about 160 pages of text). It also has the great advantage that the disks are removable, so by changing disks, its capacity is essentially unlimited. The Applix 1616 will accept two floppy drives, and these are known as **/f0** and **/f1** (with modifications, you can add up to 8 drives).

Generally, you will need at least one such floppy or diskette drive to make any serious use of your Applix 1616. For one thing, you can not really run 1616/OS v4.x satisfactorily without a disk drive (although everything described in this manual so far will work). For another, virtually all the programs available for the Applix 1616 are provided on 9cm disks, not on cassette.

Before you can use a brand new diskette you must both *format* and *initialise* it. This process places magnetic 'tracks' on the diskette, onto which files and programs can be stored. Formatting and initialising a disk totally destroys the previous contents, so **do not** format a disk on which you have programs. You format a disk using a utility program called **`blockdev`** (this replaces a much earlier version called **`ssddutil`**). This program is supplied to you on your 1616/OS Version 3 or 4 user disk, and is usually in directory **`/bin`**. It is menu driven, and can also convert old Version 2.4 disks so they can be read by Version 3 or 4. You invoke it by typing `blockdev /f0`, or `blockdev /f1`, depending upon which drive is to be used.

The largest capacity disk device available is the hard, or fixed, disk, known as **/h0**, **/h1**. These are much faster than floppy drives, nearly as fast as the ram disk. They also have much greater capacity. The smallest is at least 20 megabyte, or 20,000k bytes, but you can get them 20 times as large. You can store at least 5,000 pages of text on a hard disk.

One disadvantage of the hard disk is that it is permanently sealed, and you can never remove the disk. Once you run out of space on it, you will have to delete something to make room for the next. Of course, those who haven't used a hard disk find it unlikely that they will fill one ... those who use them tend to fill them with remarkable speed. The moral is, get the biggest one you can afford!

The other major disadvantage of the SCSI hard disk is the cost. At the moment, you can expect to pay between $400 and $1500 or more, depending on the size.

Hierarchical file systems

Version 3.0 and higher of 1616/OS store their files in a hierarchical or tree structure. With large capacity disks, the keeping of all a disk's files within one directory becomes unmanageable; sub-directories are used to categorise disk files, as explained below.

Directories

A disk directory is a list of information about a number of disk files. Each section in the list is called a 'directory entry'. A directory entry may describe either a file, or another directory.

By putting directories within directories, a tree-structured file system is built. Because the structure is like a tree, the main directory is generally known as the **root** directory, and for historic reasons is shown as '/'. (Experienced computer users should note that MS-DOS uses a backwards slash '\' instead.)

There is no limit to the number of files or directories that can be contained in the root directory, or any other directory. When a directory is first made, it contains space for 16 entries. It expands by an extra 16 entries as required. If it contains too many entries, it tends to slow down disk access. However you can make other named directories, each of which also can contain multiple names. Each of these directories can also contain named directories, and so on!

In general, directories that contain more than about 50 to 100 files tend to get unwieldy.

You can create a new directory using the `mkdir pathname [pathname ...]` command. You should try to make the 'pathname' both meaningful, and short (if you make it too long, you will get tired of typing it). You can create several new directories simultaneously with this command.

You can move into the new directory using the '`cd` pathname' command. You will notice that the command prompt will change from something like **/f0/bin** to show which sub directory you are now in. If you now create another directory using `mkdir`, and move to it using `cd`, your command prompt will show both directory names. You can suppress the display of the directory path by using the `option` command (see *Reference* section of this manual for details).

You can move back to the previous, or parent, directory using '`cd ..`', and continue doing this until you reach the root directory. You can also reach the root directory directly, by typing the shortcut `cd /`. The directory you are currently in is, for historic reasons, known as '**.**'

Filenames and pathnames

As we mentioned, each file and sub-directory has its own name. This name may be up to 31 characters long and may not contain any '/' characters. This is the **filename**. A **filename** is a string of characters which does not contain any directory specifications, such as '`ssasm.xrel`'.

A **pathname** includes a directory specification, such as "`/rd/myfile`", "`bin/edit.xrel`" and "`../test`". A pathname may include the name of the disk device, such as **/rd**, or **/f0**, or **/h0**. It may be be merely the name of some directory, such as **/bin** or **../test**. It may include a filename, as in **bin/edit.xrel**. It may include disk, directory and filename, as in **/f0/bin/edit.xrel**. Notice that each directory name is separated by the delimiter of a / character.

A **full pathname** starts with a disk device, /rd, /f0, /h0 or similar, and lists the entire pathname, starting at the root directory. However, there is often an easier way to reach a particular sub-directory, by starting from whichever directory you are currently in.

When a pathname does not start with a '/device' name it is a **relative pathname**: the full pathname is obtained by concatenating the current directory pathname to the relative pathname. For example, you can go back one parent directory using `cd` **..**, or back two using `cd` **../..**, or move back and up to another directory using something like `cd` **..**/newdir/topdir. You can specify that you are starting a relative path from your current directory by indicating your current directory as '**.**'

Examples:

If the current working directory is "/RD/BIN/SUBDIR"

Relative pathname	Full pathname
..	/rd/bin
../myfile	/rd/bin/myfile
.	/rd/bin/subdir
../../test.s	/rd/test.s
sub2/myfile	/rd/bin/subdir/sub2/myfile
./textfile	/rd/bin/subdir/textfile

Typical file commands

- The names of all your files are held in a 'directory', and you can see a very detailed list of the names of your files by typing `dir`. A directory listing includes various file attributes (detailed later), as well as the size of files, and the date they were created. Each file takes a line of the display. You can use wildcards (the * symbol, used as `dir *`) to include sub-directories, or restrict the range of the directory command by doing something like `dir bi*`, which would show all files and directories starting with `bi`.

- If you want a shorter list, use `dirs`, which stands for 'directory short'. This lists the filenames only, in multiple columns across the screen. As with the normal directory, files are normally listed in alphabetical order, although you can alter this using the inbuilt `option` command. Directories are listed first, with a / preceding them, to indicate they are directories.
- Files can be `deleted`, which means they are removed from the directory. Directories can also be `deleted`, provided they contain no files. Wildcards can be used, or multiple filenames specified. For example, `delete letter1 junkmail stuff.xrel` would delete the three files named.
- Files and directories can be `renamed`, which merely changes what they are called. Typical commands include `rename geg57 geg58`, which renames the directory `/geg57` so that it becomes `/geg58` Notice the / in the directory listing, indicating we are dealing with a directory full of files, not an individual file. The same commands work with files or directories.
- You can `copy` a file, or files, or directory, thus having them in two different places, using a command such as `copy *.xrel /rd/bin`. This would copy all files ending with `.xrel` in the current directory into the `/rd/bin` directory (assuming that existed), under their present names. You can also change the name of files as you copy them, by a command such as `copy abc cba`. Using a wildcard, such as `copy abc* cba`, is a mistake, because there would be more than one filename to change, but only one destination. The system will complain that the destination is not a directory in this case.
- A slightly subtle difference is to `move` a file or directory, usually to a different directory. Moving a file or directory means that only one copy of it exists, rather than two copies.
- You can display the contents of a file with `type`, `cat`, or `cio` (there is often more than one way to do similar things with files, in this case `type` is for MS-DOS users, while `cat` is for UNIX users, and both actually use `cio` to do their dirtywork). Since wildcards are accepted, `type *.c` will show the contents of all `.c` files in a directory (`type` will also show the filename before displaying each file if there is more than one file). For obvious reasons, `type` complains if you want to see a directory.
- You can update the time stamped on a file or directory using `touch`. It brings all the files specified up to today's date, without otherwise changing them. Handy for programmers, since many compilers are fussy about dates and times. As always, all wildcards are accepted.
- To help prevent accidental destruction of a file, and to keep track of when it was last backed up (saved elsewhere), you can change its lock and backup 'attributes' using `filemode`. We will cover this and other more advanced commands later.
- Since the computer only looks in your current directory (the one that appears in your present prompt) for a file, you can tell it to search other directories or even other disks as well by using the `xpath` command.

All these, and other, disk commands are described in detail in the reference section of this manual.

5
Starting Programs

This section continues the discussion of commands, by expanding upon the treatment of "disk" files by 1616/OS. Obviously, the concepts here will be of most use when you start using programs that use the disk drives. The chapter includes starting a program, the different types of programs (`.exec` and `.xrel`), and discussion of using shell files.

Starting a program

Starting a program is easy: you type its name, leaving off the extension. Programs can be identified by their extension, which is always `.xrel`, pronounced "dot exrel". Older programs are indentified by `.exec`.

Search path

When the name of an executable program is typed in on the 1616/OS command line, and it does not refer to an executable memory-resident driver or an inbuilt command, the system searches for a disk file to load and execute.

- It first searches your **current** directory for a file whose name is the command you typed with '`.xrel`' added.
- It then searches for a file with '`.shell`' added;
- If no '`.shell`' file is found it searches for a file with '`.exec`' added.

As mentioned, the **current** directory is the one that appears in your prompt. If you have used the `xpath` command correctly, additional directories are searched. If none of these three programs can be found in your current directory, then searching for these three files continues in other directories. The directories that are searched are as specified by your current execution search path, which is set using `xpath`. You can check for the current execution search path at any time simply by typing `xpath`.

You specify which other directories are to be searched by using the `xpath` command. In this manner, frequently used files can always be found, even when you are in some other directory. You should not include unwanted directories in your `xpath` commands, as this simply slows down access to programs. Typically, you would include the command `xpath /f0/bin /f0/utility` or some similar sequence soon after starting the system. As an alternative, you can easily add directories to the search path by using `xpath + /f0/newpath` or similar at any time. Take care with `xpath -`, which removes all search paths.

Types of executable files

A file whose name ends in '`.xrel`' is a relocatable MC68000 machine language program. It is designed to be loaded into memory and executed.

A file whose name ends in '`.shell`' is a text file (created with `edit`) containing a list of commands which 1616/OS interprets when the shell file is executed.

A '`.exec`' file is non-relocatable MC68000 machine language. It is loaded at a specific, fixed address in memory for execution. If this particular memory is not free the `exec` program cannot be executed, so you should only use `.exec` programs sparingly. They are really an obsolete form of program.

In general `xrel` files are preferable to `exec` files because multiple programs can reside in memory when they are relocatable. `exec` files are used during program development and debugging or for special applications; they are partially a relic from earlier versions of 1616/OS, and may not be available in future versions. Please note that you must have Version 3 or later of the operating system to use `xrel` files, and that almost all programs produced are in `xrel` format.

Executing binary files

When you execute a binary program from an `.xrel` or `.exec` file it is loaded into memory and executed. If it is an `exec` file it is loaded at the start address indicated by its 'load address' attribute (the third column in the 'dir' display). If it is an `xrel` file it is loaded at the highest available address where it will fit. If the 'load address' is unsuitable, or the program will not fit into available memory, an error message is displayed and the program is not run.

The contents of the command line are passed to the loaded program so that you may supply it with any options, file names, etcetera which it may need. This is how 'arguments' are passed along to programs, and why most programs are written to accept command line arguments.

If the program is correctly written, then any output which it normally produces upon the screen may be redirected *to* devices or disk files by using the '>' or '>>' construct on the command line. Any keyboard input may be obtained *from* a device or a disk file by using the '<' construct.

Error messages or other special output may be redirected *to* devices or files with the '}' or '}}' constructs.

When the program has ended it returns control to 1616/OS's command interpreter. There is much more about this topic in the *Programmer's Manual*, the *Technical Reference Manual,* and the *Assembler Manual*.

Executing shell program files

When you execute a `.shell` file, by typing its name (without the `.shell` extension), lines from it are read and executed as if they had been typed from the keyboard. Control is returned to the keyboard when the file is exhausted.

Comments may be put in `.shell` files. They are preceded by a semicolon.

Any arguments following the command file name are substituted into special symbols within the lines before they are executed. The special symbol '**$1**' refers to the first argument, '**$2**' to the second, etc. The '**$0**' symbol returns the name of the `shell` file itself. The symbol '**$***' refers to all arguments except the name of the `shell` file (arguments 1 and on). Two very simple and silly examples of `.shell` files:

```
; A shell program to rename a file
; Usage:  ren oldname newname
rename $1 $2   ; Pass the arguments on to the OS
```

```
; A shell program to delete files like MS-DOS!
; Usage: del file1 file2 file3 ....
delete $*     ; Delete all the files specified
```

You should carefully examine the various shell files on your User disk, and on the various Applix freeware disks for more extensive 'real' examples of how to use these files. If readers pester me about this, I'll find some better examples.

Shell file error trapping and command echoing

There are five commands which are only allowed within `shell` programs. The commands `trap` and `notrap` enable and disable error trapping mode. The commands '`+`' and '`-`' enable and disable command echoing mode. If no commands are specified then error trapping and echoing both default to the disabled state.

Error trapping mode

If a `shell` program is executing in error trapping mode (by placing a `trap` command early in the file), and it executes an inbuilt or transient command which flags an error, then the `shell` program will be terminated. Errors are indicated by returning a negative number in the MC68000's d0 register, as described in the *Programmer's Manual*. All inbuilt commands flag errors if something goes wrong. Correctly written transient programs should return an error flag if they detect some form of failure.

The command `trap 2` in a shell file causes it to exit if a command returns **any** non-zero exit status. This differs from the normal `trap` command, which only traps a negative exit code. The new addition to Version 4 is designed for programs ported from UNIX, where error exit codes are anything non-zero. The HiTech C compiler passes normally return non-negative exit codes, so it is of use here.

In non-error trapping mode, errors are ignored (but the rest of that line is not used) and the next line from the `shell` file is executed.

Echoing mode

If echoing mode is enabled (by placing a **+** early in the file), then each line from the `shell` file is printed out with the character string '---->' in front of it before it is executed. Use this for testing how a shell file works, or for informing a user (via display on the screen) that things are happening. If echo mode is off, then the only output which occurs is that produced by the commands which the `shell` file invokes.

Multiple or nested shell programs

The execution of `shell` files is fully nestable: `Shell` files may be called from within `shell` files. Parameters may be passed from one `shell` file to the next with the $1, $2 mechanism. The depth of nesting of `shell` files is limited by the maximum number of disk files which can be open simultaneously, generally sixteen.

When a `shell` file program is invoked by another `shell` file, the trapping and echoing status of the calling `shell` program are not normally transferred to the called one. If you desire trapping and echoing in the called `shell` program you will have to put '+' and 'trap' commands in it.

If a `shell` program which was invoked by another `shell` program is in error trapping mode, and an error is detected, the error flag will be returned to the calling `shell` program. The calling `shell` program can then abort, if it is in trapping mode.

<< redirection

When the shell file interpreter encounters a line such as

```
command args ...        <<eofmarker
<stuff>
<more stuff>
eofmarker
```

it asynchronously executes the first command with its standard input attached to a pipe. The data up to the line *eofmarker* is fed down the command's normal input, after which the normal interpretation of the shell file continues.

The opening *eofmarker* must be all UPPER CASE characters in the range A to Z only. There must be no trailing white space after the *eofmarker*, and no space between the << and the *eofmarker*.

The closing marker must be identical to the opening marker, and must be on a line of its own. If it has a **&** character added to it, the shell interpreter will not wait

for the command to terminate, but will continue to interpret the shell file from the closing marker. This is great for feeding lengthy information into commands or programs. Again, if readers pester me, I'll come up with some lengthy examples.

Shell file example

Almost any non-trivial example will be hard to understand for a beginner. I suggest that you start by taking any 'long' command line (or sequence of command lines) you happen to use, and type that in as a shell file.

As an example of `shell` file programming, suppose that we wish to write a `shell` program file which, with a single command, will allow us to assemble an assembler source code file. We wish to produce a listing file, delete its editor backup file, perform a directory listing of the file and then edit the file. An appropriate `shell` file would be:

```
;Shell file to perform assembly functions
;Usage: doasm asmfilename listfilename
;
+                  ;Echo commands to screen
trap               ;Stop on error
SSASM -l $1.s >$2.lst ;Perform assembly
notrap             ;Prevent shell exit
                   ;if backup file is not present
delete $1.bak      ;Remove the backup file
dir $1             ;List the new directory entry
pause .100         ;Wait two seconds
edit $1.s          ;Edit the source code
```

This file may be created with the 1616/OS editor by entering `edit doasm.shell` and typing the text. And again, if readers pester me, I'll come up with some longer examples.

6
Edit: The Screen Editor

The inbuilt editor is a convenient method of creating text files. The 1616/OS full screen editor is designed for editing text files, for use as programs. Its commands are similar to those of MicroPro's WordStar word processing program in 'non-document' mode.

The 1616/OS editor uses many of the features of the 1616 environment. It works in any of the video modes (320 or 640 columns, or MGR windows) and in text windows of any size. It may be used by application programs which run under 1616/OS. For example, if you are writing a program, such as a text formatter, or data base, that needs extensive typing, you can have your program call upon the full screen editor for accepting the typing. In other words, it is very much more elaborate than the MS-DOS editor `edlin`, so you should learn how to use it correctly.

Using an editor

EDIT exists either as an inbuilt command in the 1616/OS ROMs or as a `.xrel` transient program. It is present as an inbuilt command in all 27512 based systems (that is, Version 3 and up). As the inbuilt version can only be invoked once while multitasking, it is worthwhile to keep a copy of the `edit.xrel` disk version available for those times when you may want multiple copies of the editor operating simultaneously. The disk based version can also be used by those with Version 2 or earlier of 1616/OS.

A much more elaborate version of the editor, *Dr Doc*, with full on screen formatting, and printer support, can be obtained on disk from Applix for $29.95. This enlarged version unfortunately will not fit in EPROM.

The cursor

Throughout the description of the editor the term 'the current line' refers to the line upon which the 1616's cursor appears. The number of this line is displayed on the editor status line.

The flashing cursor represents your current working position within the file, and almost all commands are effective at the cursor's position. In the case of text insertion (single character, block move or copy and file read) the current position may be thought of as being in between the character under the cursor and the one preceding it.

Starting the editor

The editor is started by typing `edit filename`. If the file does not exist then `edit` creates a new one. If you enter a number after the filename then this number

becomes the tab column width - this is a useful feature for program files which are heavily indented with tabs. If the editor is supplied on disk, you must have the correct disk in your drive, and the editor must be in the current directory, or in a directory that is in your search path.

When the editor is started it determines the size of the file and allocates for itself sufficient memory space for the file, plus 32k (kilobyte, enough for about an extra 30,000 characters of text). The size of the edit buffer is displayed as the file to be edited is read in. Because of this memory allocation scheme you cannot increase the size of the file by more than 32,000 characters during a single edit session; if you get the 'file too large' message on the status line then you will have to write the file out (using the Ctrl K X command) and re-edit it. This isn't usually much of a problem.

General operation

The editor works on the basis of always displaying all of the current line; partial lines are displayed but if the cursor is moved to a line which is not fully on the screen then the display is scrolled to bring the line into view.

A 'marker' is a position within your text file which you supply to the editor and which is remembered by the editor. The markers are used for remembering positions in the file. If text is added or deleted before, between or after markers they stay at the same position in the text, so that if you place a marker at the start of a particular word then it will stay at the start of the word if text is inserted or deleted before it. The positions of markers are lost if you quit from the editor (using the Ctrl K Q or Ctrl K X commands).

Whenever the editor reads or writes a disk file the user is first prompted to enter the filename. The editor makes a guess at the name of the file and if it is correct you need only press the Enter key; otherwise you may edit the filename in the usual manner before pressing Enter.

Whenever `edit` prompts for an output filename, placing a **>** character at the front of the filename results in the text to be written being appended to that file, rather than overwriting it as is normal. This feature is only available from 1616/OS Version 4 and up.

Once within the editor a status line appears at the top of the screen. From left to right, the status line consists of:

- A general message display area for status information and command echoing.
- The cursor position within the file, measured in characters from the start of the file
- The current line number
- The current character number within the current line
- The name of the file which is being edited

Entering text

Enter text into the file by positioning the cursor at the desired place and simply typing. There is no 'insert' mode. All non-control characters go directly into the file. In the expanded *Dr Doc* disk version, you can toggle an *overwrite* mode by using the Ins key.

You may insert any control character except a Ctrl M into the text by preceding it with a Ctrl P, as with the line editor. This is useful for inserting special escape sequences such as those which start boldfacing and underlining on your printer when the file is printed. For serious text formatting, and full printer support, we suggest that you use the enhanced editor *Dr Doc* available on disk from Applix for $29.95.

Editor commands

All of the editor's commands are either a single control character or a single character preceded by a control-Q or a control-K. The 1616's numeric keypad cursor control keys are programmed to generate codes which are suitable for use with the editor. The editor commands can be broken up into the following groups:

Cursor movement	The cursor movement commands allow you to display and/or alter different parts of the text file by moving to different positions within it.
Scrolling	The scroll commands move the screen display without altering the relative cursor position. Handy when you are used to them.
Text deletion	There are a number of commands for deleting ranges of text before or after the cursor.
Block commands	Blocks of text may be marked and manipulated.
File commands	Various file I/O commands and system access commands are available from within the editor.
Miscellaneous	The miscellaneous commands include setting and moving to file markers, merging files, partial screen freezing, etc.

Editor commands are described in detail below; a quick reference chart is given in appendix A.

The **Control-Q** and **Control-K** commands are two keystroke commands. You must first type the Ctrl Q or Ctrl K and then the selected letter. For example, the command Ctrl Q C is entered by typing a Ctrl Q in the normal manner, followed by a c or a C or a Ctrl C. The 5 key on the numeric keypad can optionally generate a Ctrl Q sequence if the number lock is off.

Cursor movement commands

The cursor movement commands are based upon file lines (groups of text separated by Enter or Return characters) rather than display lines, which means that a movement from one line to the next may involve the cursor moving vertically by more than one display line to preserve its relative horizontal offset.

The editor attempts to keep the cursor at the same horizontal position during vertical moves. This is good for program files because they have short lines, the length of which tends to vary considerably. If you are editing files which have lines which extend over several display lines (ie., normal text, not programs) you will find that the line up and line down commands, Ctrl E and Ctrl X are not all that you might want. Use the Ctrl Q S and Ctrl Q D commands instead.

Note that some of the block and marker commands (see below) are also used to alter the cursor position.

Ctrl E, ↑	Moves the cursor to the same column in the preceding line.
Ctrl X, ↓	Moves the cursor to the same column in the next line.
Ctrl Q E	Moves the cursor to the top of the screen.
Ctrl Q X	Moves the cursor to the bottom of the screen.
Ctrl R	Moves up the file about 2/3 of a screen.
Ctrl C	Moves down the file about 2/3 of a screen.
Ctrl Q R	Go to the first line in the file.
Ctrl Q C	Go to the last line in the file.
Ctrl D, →	Go forward one character.
Ctrl S ←	Go backward one character.
Ctrl F	Go to the start of the next word.
Ctrl A	Go to the start of the previous word.
Ctrl Q D	Go forward 80 characters within the current line.
Ctrl Q S	Go backward 80 characters within the current line.
Ctrl B	Go to the start of the current line. If at the start, go to the end.
Ctrl J	Go to the start of the next line.

The Ctrl A and Ctrl F commands make assumptions about which characters are used to separate words. The separators at which these commands stop are a space, a tab, a newline or one of the following characters:

. ! , : ; ? ( ) [ ] { }

A good way to help memorise these commands is to think of the ‘e’ key as the top of a diamond pattern that includes the ‘s’ (left), ‘d’ (right), and ‘x’ (down). The ‘a’ key moves left a word at a time, and the ‘f’ key to the right, a word at a time.

You can redraw the display at any time by pressing Esc. This is provided because other programs running simultaneously in background may put output on the display when it is not wanted.

Scrolling commands

The scrolling commands roll the screen up or down by one display line. The cursor is left in the same position within the file, so it will move within the display. If the cursor is pushed off the screen by the scroll command then it is moved back into view in an appropriate manner. The scroll commands are useful for displaying a line which is just beyond the screen limits with few keystrokes.

Ctrl Z	Scroll the display up a line.
Ctrl W	Scroll the display down a line.

Text deleting commands

The text deleting commands remove text from your file. You cannot 'undo' a delete command as you can with the line editor. Note that the block command Ctrl K Y also deletes text.

Ctrl G, Del	Grab, or delete the character under the cursor.
Ctrl H, Bs	Delete one character backward (the last character typed).
Ctrl T	Delete from the cursor to the start of the next word.
Ctrl Y	Delete the entire current line.
Ctrl Q Y	Delete from the cursor to the end of the current line.
Ctrl V	Delete from the start of the current line to the cursor.

The Ctrl T command uses the same word terminators as the Ctrl A and Ctrl F cursor movement commands above.

Undo commands

Whenever any text is deleted, it is placed in one of ten *undo* buffers. The undo buffers can be reviewed with Ctrl Q U. Typing Ctrl U 0 to Ctrl U 9 inserts the contents of an undo buffer at the cursor. Ctrl U U is shorthand for Ctrl U 0. The undo buffers work on a last-in first-out basis; when some text is deleted from the file, the oldest entry in the undo buffer is removed to make space for the new entry. This undo facility is new in Version 4.

Block commands

With the block commands you mark out a block of text and then move it, delete it, copy it or write it out to disk. The block is marked out by putting the cursor onto the first character of the block and entering Ctrl K B; the cursor is then put at the next character beyond the end of the block and Ctrl K K is entered. These two operations may be reversed, but the beginning marker (Ctrl K B) must always be closer to the front of the file than the end marker (Ctrl K K). When both markers are correctly entered the marked block is re-displayed in a different colour, or with

a different brightness on a monochrome monitor. You may move the beginning or end marker simply by repositioning the cursor and placing the marker again; the screen will be updated appropriately.

If you make an error in using the block commands a message is printed on the status line. Common errors are attempting to move a block to some position within itself and making an error setting the markers.

It may be a good idea to use the Ctrl K D command to write the current file out to disk before using some of the block commands: they can be destructive!

Ctrl K B — A beginning marker is put at the current cursor position.

Ctrl K K — Put the end marker at the current cursor position.

Ctrl K Y — Delete all the text between the beginning and end markers.

Ctrl K V — Move the marked text to the current cursor position. The marked block is removed from its original position and placed in the new one.

Ctrl K C — Copy marked block to the current cursor position. The marked text is left where it is and it is also copied to the new position.

Ctrl K P — Put the currently marked block into the head of the undo buffers.

Ctrl K W — Writes the marked text out to a disk file. The user is prompted to enter the filename.

Ctrl K H — Hides the marked block. The block becomes inaccessible and the highlighting is removed. Typing Ctrl K H again restores the block.

Ctrl Q B — Go to the start of a marked block.

Ctrl Q K — Go to the end of a marked block.

File commands

Ctrl K R — Reads an entire disk file into the file which you are editing. The new text is placed immediately in front of the current cursor position. You are prompted to enter the name of the file to be read.

Ctrl K D — Disk write. Writes the current file out to disk and resumes editing at the previous position. All markers and the search and substitute patterns (see below) are preserved. This is a useful command for making a periodic backup of a file. You are prompted for the name of the output file.

Ctrl K X — E**x**it. The normal way of quitting from the editor. You are prompted for a filename, the edited text is saved and you are returned to the 1616/OS command level.

Both the Ctrl K D and the Ctrl K X commands preserve the file in its original form (before editing) in a `.bak` file. This is for safety purposes: if something went wrong with your last edit then you still have the original file. You can stop automatic production of a `.bak` file by using `option .15 1`. See the `option` command in the *Reference Section* for full details of all options.

If reading from a file in which an EOF (end of file) character accidently appears, the standard input reverts to device TTY: This essentially means that 1616/OS makes an inspired guess about where to look for its next command; handy if working the computer remotely, say over a phone. Under normal circumstances, you won't notice this feature.

Miscellaneous commands

The miscellaneous commands cover various odds and ends which you expect to have in a text editor, and then some.

Ctrl K Q — Quit from the editor without saving the file to disk. You would do this if you had made some changes which you later decided to reverse, or if you made a serious editing mistake, or if you were simply using the editor to browse through a file. If the file has been altered since it was last written to or read from disk then the message 'Not saved yet' appears on the status line and you must enter the Ctrl K Q command a second time to quit. This is a safety feature which prevents you from accidently quitting before saving your work.

Ctrl K E — Execute one 1616/OS inbuilt or transient command. A prompt is displayed and you can type the command. Upon completion of the command you are prompted to press the Enter key and then editing is resumed as before.

Ctrl K I — Temporarily escape from the editor and execute 1616/OS inbuilt and transient commands. The 1616/OS command line prompt gets another '>' character to remind you that you are still within the editor. To return to your file, use the 1616/OS 'quit' command, or type the end-of-file character (usually Ctrl D).

The Ctrl K E and Ctrl K I commands are temporary escapes from the editor. When you return to the editor all the block markers, the substitute pattern and the search pattern are undisturbed. The only 1616/OS command which you cannot use with the editor escape is 'edit' - you can only edit one file at a time. This only applies to the ROM version of the editor in 1616/OS Version 3.

Ctrl Q G — Go to a particular line within the file. You are prompted to enter a line number. Pressing Enter immediately is equivalent to entering the number 0, and sends you to the start of the file. Entering an impossibly large number (or -1) sends you to the end of the file.

Ctrl K F — Freeze part of the screen function. Often when editing a file (particularly with program source code) you wish to look at two parts of the file at once. This may be done with the screen freeze function in the following manner:

1) Use the cursor positioning commands to position one of the sections of the file which you wish to see towards the top of the screen.

2) Position the cursor under the text in which you are interested, so that the text lies between the top of the screen and the cursor row.

3) Type Ctrl K F. This freezes the screen above the cursor. All editing now occurs under the divider line which is drawn. You may now move about the file within this smaller screen window.

When you no longer require the frozen display, restore the normal display by typing Ctrl K F again. The editor will not allow you to freeze too closely to the bottom of the display, as there must be sufficient active screen area left for the editor to work in.

Ctrl K 0-9 Places a marker within the file for you to return to later. Type Ctrl K followed by a single digit number (0 - 9). When moving about a large file it is often convenient to place a marker at your current position, move somewhere else for a while and then jump back to your original place. You may place up to 10 markers within a file; the markers are lost when you quit from the editor - they are not actually part of the file.

Ctrl Q 0-9 Moves the current position to the marker corresponding to that digit (see Ctrl K 0-Ctrl K 9, above), used by typing Ctrl Q followed by a single digit number. An error message is displayed if the place marker has never been set.

Ctrl Q F Find text, or search. You are prompted to enter a pattern which the editor is to find in the file. The editor also asks for the number of repetitions of the search: if you ask for 10 repetitions then the cursor will be moved to the tenth occurrence of the pattern within your file. If you simply press the Enter key when asked for the number of repetitions the editor will find the next occurrence of the pattern. The search starts from the current position and proceeds forwards. The number of occurrences actually found is displayed on the status line; the cursor is left on the last found occurrence.

Ctrl L Last search repeat. Finds the next occurrence of the pattern which was searched for in the most recent Ctrl Q F pattern search command.

Ctrl Q A Replace (substitute) text. The editor asks for a pattern for which to search; when this has been entered the editor asks for the characters which must replace the original pattern; it then asks for the number of times which the replacement must be performed. If you press the Enter key in response to the 'Repetitions?' question then the replacement is done only once. Entering an impossibly large number

in response to this question makes the replacement effective through to the end of the file. The replacing is done from the current position forwards.

Ctrl N — Next replacement. Provides a single repetition of the previous Ctrl Q A command. The search is performed from the current position.

When entering the search and substitute character strings for the Ctrl Q F and Ctrl Q A commands the control-N character (Ctrl N) may be used to match with the new-line character in a file. If, for example, you wished to find the next line which started with the letter 'T' then you would search for the pattern 'Ctrl N T', which represents a new-line followed by a 'T'; this search would not match with 'T's which are not the first letter on the line.

The substitute command Ctrl Q A may be used to remove selected patterns from the file by immediately pressing Enter in response to the 'With?'question; the selected pattern is searched for and is replaced with an empty character string, effectively deleting it.

Editor hints

- If you are altering a file and find yourself typing a sequence of characters and/or commands over and again, escape to the 1616/OS `shell` and program the characters and commands into a function key and use it instead. See the 'fkey' inbuilt command below for a description of programming function keys. You can put any character into a function key definition, including the control characters which the editor uses for commands. You can also use keyboard programming via the Alt Ctrl function key method.
- Occasionally write your file out to disk with the Ctrl K D command; if something regrettable occurs you will at least have a recent copy.
- Learn all the features of the editor at an early stage. If you limp along using only half of the available commands you will regret the wasted time when you finally come to learning all of them.
- Use the Ctrl K 0-Ctrl K 9 place markers and the partial screen freeze feature, particularly if you are editing program source code. They're great.
- The Ctrl Q F search command can be used for counting the number of occurrences of a pattern within a file: position the cursor at the start of the file and search for an enormous number of occurrences; the editor will display the actual count on the status line when it has finished. Search for spaces for a rough word count.
- To move a single line, use Ctrl Y to delete it, move the cursor to where it should be inserted, then undo the deletion using Ctrl U U.

- If running `edit` over a slow serial line, you can optionally eliminate the periods (.) at the end of the file and in the status line. Specify lowbaud in your environment.
- You can use redirection to send a set of commands to the editor, so you can actually edit a set of files by "remote control". Use this trick when you have standard changes to make to a lot of files. See the Applix demo disk for an elaborate example.

Fancy text

If you embed terminal control characters or escape sequences in a text, you can display **bold**, underline, *italics*, $_{\text{subscript}}$, and $^{\text{superscript}}$ text (or any reasonable mixture). The $29.95 *Dr Doc* editor uses these extensively, and also will allow them to be printed. However, any 1616/OS command that writes text to the display (`cat, type`) will also work. These sequences are displayed or used whenever the Applix 1616 encounters them, thus making it relatively easy to employ fancy text in your text files.

Test them by using Ctrl P to embed an Esc key in your text. Thus, from the keyboard, `echo` Ctrl P Esc G 4 will put the display into **bold** mode, etc. Listed below to star in your text are the main sequences that produce text attributes. The full list of escape sequences is in the 1616/OS *Reference Section* at the end of this manual, after the `term` entry.

^G	Beep speaker 7.
^L	Clear screen 12.
ESC)	Start highlighting.
ESC (	End highlighting.
ESC *	Clear the screen, or current window.
ESC B	(value+32) Sets the background colour to ‘value’.
ESC b	Visible bell.
ESC F	(value+32) Sets the foreground colour to ‘value’.
ESC G 1	Sets subscript mode.
ESC G 2	Sets superscript mode.
ESC G 4	Sets bold mode.
ESC G 8	Sets underline mode.
ESC G @	Sets italic mode.
ESC G 0	Clears subscript, superscript, underline, bold and italic modes.
ESC S	(value+32) Sets the border colour to ‘value’.

7
Multi Tasking

The operating system includes pre-emptive multitasking with pipes and signals for interprocess communication. Programs can readily and automatically pass input and output among themselves, even if they are all operating simultaneously. Disks can be accessed by multiple programs at once, without damage.

Multitasking is a highly desirable characteristic in an operating system. It simply means that it is able to run more than one program at once. It does this by 'sharing' the central processor among the multiple programs. If all goes correctly, this sharing is sufficiently fast that everything appears to be running simultaneously.

There are, of course, some disadvantages. Any individual program will run slower, since it is sharing its time. Also, it is not appropriate for programs which demand keyboard input, and produce rapid display updates, such as games. It tends to mean that programmers must think more carefully about exactly how their programs will produce output.

However there are many tasks where you will be happy to start a program, and then do something else until the results are obtained. For example, you can start assembling or compiling a program in background (since this can take a minute or so), and continue editing a file. Or you can start a print out in background, and play a game while things are printing. Another typical use is running a serial terminal in background.

By use of the `vcon` utility programs on the Users Disk, you can obtain multiple 'virtual consoles', so several programs can each display their output simultaneously. You see either a number of small 'windows' in which the various outputs appear, or else swap between several full size displays whenever you wish.

The only home computer systems that include full pre-emptive multitasking are the Applix 1616 and the Commodore Amiga, and computers running expensive operating systems such as OS-9 or Unix. MS-DOS machines, the Atari ST and the traditional Apple Macintosh do not provide multitasking, although you can sometimes obtain a limited form of context switching (changing quickly from one task to another) in each by using additional programs from other companies.

Multitasking introduced

With much compatibility the system now supports multitasking, the full bottle, quite transparent to programs running under it. This means that you can run more than one program simultaneously. Most programs devised for earlier versions of 1616/OS will still run, but some may need alterations to their stack size (use the `chmem` utility provided on your User Disk). Others (more rarely) may require changes to the source, and recompiling.

First, some terminology, much of which is standard for modern multitasking operating systems such as UNIX and Minix.

Processes

A process is one of the one or more sections of program between which the operating system is dividing the 1616's time. All the currently known processes are described by a table internal to 1616/OS. This table is called the 'process table'. The current processes can be inspected using the inbuilt command `ps`, which also shows the status of many of the multitasking operations described below.

Scheduling

A process is said to be 'scheduled' when the operating system has decided to run it for a while. When the system decides that the process has run for long enough, it is descheduled and another process is scheduled. The amount of time a process can obtain can be varied by the ***nice*** entry to the ***proccntl*** system call

Parent and child processes

When one process starts (or spawns) another, the second process is referred to as a child of the first. The first is the parent process. A typical example of this is in running shell files. The shell file interpreter in 1616/OS is the parent process, creating a child process for each command in the shell file.

Killing

A process is said to be killed when another process (or the system) causes it to terminate abnormally. This is done by sending it a signal or by explicitly killing it with the ***kill*** entry to the ***proccntl*** system call. A normal process exit is very similar to killing, except a process does it to itself and the exit code is handled differently.

Blocking

A process is said to be blocked if it is descheduled until an event happens, typically the termination of another process. The most common case of blocking is where one program runs another one, then blocks until the second one completes and exits.

Locking in a process

A process is 'locked in' if the scheduler is prevented from descheduling the process and running another one. The ***lockin*** entry to the ***proccntl*** system call is provided to permit a program to lock itself in.

Sleeping

A process is said to be sleeping if it is not being rescheduled. It still occupies a slot in the process table, and will be reactivated at some time in the future. Sleeping processes take practically no CPU time (0.1%, actually). ***Sleep*** is an entry in the ***proccntl*** system call.

Synchronous and Asynchronous processes

A synchronous, or foreground process is one which blocks its parent. An asynchronous, or background process does not block its parent, so usually the child and parent proceed in parallel. The availability of asynchronous processes is what it is all about: the only sort of process under the earliest versions of 1616/OS were synchronous.

Background processes

A background process is one which is asynchronous from the 1616/OS command processor. It will usually produce no output unless it encounters an error condition.

PIDs

Every process, whether synchronous or asynchronous, is assigned a unique identifying number, its 'process ID', or PID. PIDs under 1616/OS range from 0 to 63. The PID (and other process information) can be listed by the inbuilt command `ps`.

Daemons

A daemon is a system related program which runs in the background. Usually it manages some aspect of the computer's behaviour or it supervises access to one of the system's resources. A typical daemon is a line printer spooler which monitors a special directory, looking for files to print out. When one is found, the daemon sends it to the printer and waits for another print job. Sending all print jobs through the daemon prevents two print jobs from being printed simultaneously. Look at the `cron` and `at` programs on the *Utility Disks* for typical examples.

Pipes

A pipe is a first-in, first-out buffer, which is managed by the operating system. A pipe has two ends: the input and the output. Characters written to the input of a pipe via a ***write*** system call appear at the output, available via a ***read*** system call.

The typical use of a pipe is in communicating between two processes. One process writes to the input of the pipe, and the other reads the output. Additional pipes may be created to provided bidirectional communication. Internally, a pipe is a 2 kbyte circular buffer. Bytes written into the pipe go onto the head of the buffer. Bytes read are taken from the tail. Reading from an empty pipe or writing to a full one causes a process to be blocked. A new ***pipe*** syscall is provided. A pipe is produced at the command line by using the **|** character between commands.

Signals

A signal is a mechanism by which one process may send a small message to another one. When a process is signalled, the system initiates a call to a section of code within the process called a 'signal handler'. The signal handler is very similar in concept to an interrupt service routine. A process must install the address of its signal handling routine in the operating system when it starts up. If a process has not installed a signal handler, it is killed when someone attempts to signal it. The ***sigsend*** and ***sigcatch*** entries to the ***proccntl*** system call handle signals.

The multi-tasking features of 1616/OS impact on the use of the system both at the command line and at the programming level.

Using multi-tasking at the command line level

Three extra commands have been provided to assist in multitasking. These are `kill`, which stops a named process. The process status is given by `ps`, which lists all current processes. The `wait` command is provided to synchronise the completion of commands.

One extra control sequence is Alt Ctrl C, which returns control to the keyboard.

There are three new special characters. The **&** character after a command puts that program into asynchronous, or background, mode. The pipe command is produced by placing **|** between two or more commands, to feed the output from the first into the input of the next. In `assign` only, the \ backslash character makes the path that follows literal.

There are additional error codes and messages relating to the multitasking commands.

Killing programs

Typing Alt Ctrl C sends a signal to the current synchronous (foreground) process, which should then exit with a negative exit code. If all your programs are behaving correctly, Alt Ctrl C should always return control to the command line prompt.

Running programs asynchronously

You may run any 1616/OS command in the background, as an asynchronous task, by adding an **&** symbol at the end of the command line. The system will print out the process's PID, and start it running. If the process is going to produce output which you want to see, it is best to direct it to a file for later inspection. If you don't want to see the output, direct it onto the `NULL:` device. Examples:

Typing
```
SSASM myfile.s -l >myfile.lst }myfile.errs &
```

will start up the 1616 assembler, assembling `myfile.s`. The listing will go to `myfile.lst`, any error reports will go to `myfile.errs` and the entire assembly proceeds in the background, permitting you to edit something else.

typing
```
type myfile.lst >cent: &
```

will send the file to the printer, in the background, so you can use the machine while it prints. Simple as that.

If a process is started asynchronously, and its standard input is not redirected with the **<** mechanism, the system will connect standard input to the `NULL:` character device, which always returns end-of-file (error code -18). This ensures that a background process cannot compete with the foreground one for input.

The PS command

The inbuilt command `PS` may be used to display a list of all the current processes from the process table. Some facts and figures which emerge are:

PID	the process's ID number, in decimal.
PPID	the process's parent process PID.
HPID	home PID
TIME	the number of 20 millsecond time slices over which the process has run, converted to minutes and seconds.
Status	This represents the state of five flags in the process's process table entry. They are as follows:
W	Set if the process is waiting (blocked) on another.
S	Synchronous.
A	Asynchronous.
E	Exit pending: process about to depart.
B	Binary: process memory to be freed on exit.
K	Killed: someone has killed this process.
Load	The load address of the program; actually the call address passed to the ***schedprep*** system call when the process was scheduled.
Stack	The address of the process's stack area.
SP	The current value of the process's stack pointer.
PC	The value of the process's program counter when it was last descheduled.
I, O, E	The handles of the process's standard input, standard output and standard error streams.

The last field is the name of the process. This name may be used when referring to the process, for example, using `kill name`. The output of this command was altered in Version 4.2a.

The KILL command

The inbuilt command `kill -ak`*nn* may be used to send a terminate (or some other) signal to a background process. The process should then stop. Processes may be identified by their process number (in decimal, not in hex as is usual with 1616/OS inbuilt commands) or by their name, as appears in the listing from the `ps` command. If there are multiple processes with the same name, only one will be killed. The options are:

-a kill all children processes

-k unconditionally kill

-*nn* specify which signal to send, in decimal, rather than the default sigterm which is 15. See the *Technical Reference Manual* or *Programmer's Manual* for details of signals available, which generally follow UNIX conventions.

The WAIT command

The `WAIT` inbuilt command returns when the selected PID has terminated. It may be used to synchronise asynchronous commands. For example, suppose there is a background process called `download` running. We wish to report when the program has finished. A command to do this is:

```
"wait download ! echo ^G^G^G >con:" &
```

This command will suspend until the completion of `download` and then emit some beeps.

Using pipes

A pipe is created by typing vertical bars **|** between two or more commands. The operating system starts each command asynchronously, connecting the output of each command to the input of the next via an interprocess pipe. If the last command in the pipeline was run synchronously, the system blocks until it is complete.

Standard input for every command in the pipeline, except the first, comes from the standard output of the previous command. Standard output for every command, except the last, goes onto standard input for the next.

You can also pipe standard error using the **^** symbol, however this can not be done while also using the normal pipe.

The command line pipes are very similar to the pipes provided by the `pipe` memory resident driver under earlier single-tasking 1616/OS versions. The important difference is that under multi-tasking, all the specified tasks are run simultaneously, whereas the pipe MRD ran them sequentially, connecting their input and output via temporary files. This temporary file kludge is still used by MS-DOS, which explains several pipe collapses in that system.

Command line syntax

1616/OS command lines are getting quite complicated nowadays, so a review is in order.

The command line in its entirety is interpreted by the ***exec*** system call, so all the following comments apply to that call.

The following characters have a special meaning on the command line:

"	Quotes are used to remove the special meaning of all other characters
!	A separator between multiple commands
> >>	Standard output redirection
} }}	Standard error redirection
<	Standard input
|	A pipe to connect two commands
^	Pipe standard error
;	A comment separator
&	Run command asynchronously
<<	Redirect following lines until marker into input of command, within shell files only.

The above are listed in order of precedence.

Some examples:

```
dir | cio | cio
```

This performs a directory listing and passes the output through the `cio` command a couple of times.

```
ssasm myfile.s ! type myfile.s >cent: &
```

This will perform the assembly synchronously, then print the source file out as a background process.

```
ssasm myfile.s & ! type myfile.s > cent: &
```

Both the assembly and the printing proceed asynchronously.

```
"ssasm myfile.s ! type myfile.s > cent:" &
```

Here both the assembly and the printing run in the background, but the printing will not start until the assembly has completed.

The implications of multi-tasking for programs

Having multi-tasking affects a program's user interface. It is the responsibility of the programmer to attempt to ensure his or her programs can run in background, if the user so desires. In general, a program can not assume it has unlimited access to resources, particularly access to the keyboard or video. Input and output should be via the standard routines, so they can be re-directed, never direct to the hardware.

Wherever possible, a program should behave correctly, whether it is run synchronously or asynchronously. The ***isinteractive*** system call is provided for a process to tell whether or not it is running in the background. If it is a background process, it should produce no unnecessary output. It should not attempt to read from the keyboard, because that is being used for other processes.

8
1616/OS Commands

In this section the 60 1616/OS inbuilt commands are described in detail. An alphabetical list of the commands is available on the Applix 1616 by typing `help`, as shown below.

The commands are grouped by function in this manual, rather than alphabetically.

The groups are:

- File related commands.
- Directory related commands.
- Cassette tape commands.
- Memory manipulation or monitor commands.
- Command line redirection.
- Shell file commands.
- System commands.
- Handy utilities.
- Communication commands.

In this documentation the term ‘word’ refers to a 16 bit number (four hexadecimal digits); a ‘long’ is a 32 bit number (eight hex digits). A ‘$’ symbol in front of a number indicates that the number is in hexadecimal format (base 16). Do not put a ‘$’ symbol in front of a number when entering it on the 1616/OS command line.

The following notations are used:

n1, n2	The letter ‘n’ followed by a digit represents a number. Numbers are entered in binary by preceding them with a ‘%’ symbol. Decimal numbers are represented by preceding them with a ‘.’ symbol. Hexadecimal numbers are the default and need no prefix.
a1, a2	As with the n1, n2 format, except these numbers always refer to 1616 memory addresses.
pathname1, pathname2	These represent valid pathnames for files.
[ ]	An argument within square brackets is optional, or changes the function of the command which is being used.
....	A row of full stops after a command line represents optional continuation of the last argument on the line.

```
/F0>help help
ASCII   ASSIGN  BASE    CAT     CD      CIO     COPY    DATE    DELETE   DIR
DIRS    ECHO    EDIT    EXPR    FILEMODE        FKEY    GO      HELP     ITLOAD
KILL    MCMP    MDB     MDL     MDW     MFA     MFB     MFL     MFW     MKDIR
MLOAD   MMOVE   MOVE    MRDB    MRDL    MRDW    MSAVE   MSEARCH MWA     MWAZ
MWB     MWL     MWW     OPTION  PAUSE   PS      QUIT    RENAME  SERIAL  SETDATE
SREC    SSASM   SYSCALL TARCHIVE        TERMA   TERMB   TIME    TLOAD   TOUCH
TSAVE   TVERIFY TYPE    VOLUMES WAIT    XPATH
/F0>help ascii assign base cat cd cio copy
Usage: ascii
       ascii d
       ascii D
       ascii h H
Usage: assign (current assignments)
       assign - (removes all)
       assign /OLD (deletes)
       assign /OLD /NEW (assigns)
Usage: base number [number ...]
Usage: cat [pathname ...]
Usage: cd [directory]
Usage: cio [ascii_eof_code]
Usage: copy sourcefile destfile
       copy sourcefile1 [sourcefile2 ...] directory
_
```

HELP [cmdname] [cmdname] ....

The command ‘HELP’ with no arguments causes a sorted list of all of 1616/OS’s inbuilt commands to be printed out, as shown on the sample screen.

The HELP command may be used to obtain more detailed help concerning one or more particular inbuilt commands by using the name of the inbuilt command as an argument to HELP, as shown for `ascii`, `assign` and so on.

File related commands

The file related commands allow the user to manipulate files on disk (block) devices. A number of these commands work when applied to character devices such as CON: and CENT: as well as with disk files.

The COPY, MOVE, TYPE, CAT and CIO commands overlap in their functions and there are a number of ways of doing any one thing. This largely derives from the fact that COPY, MOVE and TYPE internally use CAT and CIO. Each command has its own application for specific functions.

Copying, joining, moving files

CAT [pathname1] [pathname2] [device1:] [device2:] ....

The CAT command copies and joins files and input from character devices. All of the named files are joined together in the specified order and are copied onto standard output. Often this output will be sent to another file or a character device using redirection. If no filenames or character device names are specified then this command gets input from standard input: in fact an internal call to the CIO command is made.

The CAT command copies by obtaining as much memory as possible then using all of this buffer for reading and writing data in large blocks. When copying from a character device CAT terminates the read when it receives an end-of-file character, normally a control-D. (See the `option 6 4` command for details of how to set end of file characters.)

Examples:

`CAT myfile > myfile.new`

Creates a new file called ‘myfile.new’ and copies the contents of ‘myfile’ to it. The file ‘myfile’ is undisturbed.

`CAT myfile1 myfile1 sa: myfile2 > myfile.big`

Creates a new file called ‘myfile.big’ and puts in it two copies of the contents of ‘myfile1’ followed by input from serial channel A and then one copy of the contents of ‘myfile2’. The files ‘myfile1’ and ‘myfile2’ remain undisturbed.

`CAT myfile1`

Dumps the file out to the screen. Naturally you can specify multiple file names.

`CAT con: > myfile`

Creates a new file called ‘myfile’, and copies everything you type on the keyboard into that file, without displaying it on screen. The Enter key is

treated as a carriage return only, so each line overwrites the previous one. To produce separate line, use Ctrl J followed by Ctrl M to produce a line feed and carriage return.

Copying files and data

COPY source1 [source2] ... destination

The COPY command is a smart file copier. It copies from character devices or files onto character devices, files or directories. The general format is:

COPY source1 source2 ... destination

If COPY detects an error it continues attempting to copy all of the specified files. The current copy movement is printed out as the copying proceeds; this indication of what is happening may be disabled with the `option 1 0` command.

When the source of a copy is a character device the copy continues until an end-of-file character is received. This character is normally a **Ctrl D**, but may be altered with the `option 6 4` command.

Whenever possible the COPY command preserves the modification date, name and attributes of copied files.

The permissible formats of this command are described in the following examples. A **pathname** refers to a valid 1616/OS filename. A **device** is the name of a character device, such as CON: or SA: A **directory** is the pathname of an existing 1616/OS directory.

`COPY pathname1 pathname2`

Copies a single file

`COPY pathname1 pathname2 ... directory`

Copies one or more files into a directory

`COPY device:  pathname`

Copies from the device into the named file

`COPY pathname1 pathname2 ... device:`

Copies the contents of one or more files onto a character device

`COPY pathname1 pathname2 device1: pathname3 device2:`

Copies the contents of files as well as characters from a device onto a character device

Moving files and data

MOVE source destination

The MOVE command is similar to the COPY command except that when a source file is copied to a new file the original file is removed. However a MOVE from a file to a character device does not result in the source file being deleted. A MOVE with a character device as a source behaves the same way as in a COPY, which although not precisely consistent, is what is usually desired (and is also the only possible way of handling this situation).

A special use of the MOVE command is moving directories within a file system:

```
MOVE directory pathname
```

will move the named directory to somewhere else in the file system of a disk device. This cannot be done if the source and destination pathnames do not refer to the same physical disk device.

When a file or directory is MOVEd to another place on the same disk no actual copying of data takes place: it is all done by shuffling of directory entries. This is why a move can delete sources when files or directories are involved, but not when character devices are involved.

Displaying files

TYPE pathname1 [pathname2] ....
TYPE device: [pathname1] ....

The TYPE command causes the contents of one or more files or character devices to be displayed on standard output (normally the video monitor). If verbose mode is set (see the `option 1 1` command) headers are printed out before each source is output. Any mix of files and character devices may be specified. If a character device is specified, then TYPE proceeds until an end-of-file character (normally control-D) is encountered. This is essentially a more limited version of `cat`, included for MS-DOS compatibility.

Deleting files

DELETE pathname1 [pathname2] [pathname3] ....

All of the named files and directories are deleted, making the disk space which they previously used available for reuse. Wildcards are useful (but dangerous) with this command.

Directories may be deleted in this manner. A directory can only be deleted if it is empty (contains no files or sub-directories) and is not in the path of your current directory.

Renaming files

RENAME pathname filename

The RENAME command changes the name of the file or directory identified by ‘pathname’ to ‘filename’. All other file attributes are preserved.

RENAME cannot move a file from a particular directory, so the command

```
RENAME /rd/mydir/myfile newname
```

will rename the file to /RD/MYDIR/NEWNAME, regardless of the current directory. Essentially a more limited, safer version of `move`.

Refreshing a file's date

TOUCH pathname1 [pathname2] ...

The TOUCH command changes the modification time/date in one or more files' or directories' directory entries to the current system time and date. This can be very handy when arranging backup copies of current files, or for programmers who need to keep file dates consistent.

Changing file attributes

FILEMODE 0 mask file1 [file2] ... Clear attribute bits
FILEMODE 1 mask file1 [file2] ... Set attribute bits

The FILEMODE command allows the setting and clearing of a file or directory's attribute bits. These bits are displayed as part of the directory entry, in the order `DALRWXS`.

Each disk file has 16 attribute bits in its directory entry. Ten are currently defined:

Bit	Mask	Usage
0 A	$0001	Backup bit: the file is backed up
1 D	$0002	Directory bit: the directory entry refers to a directory
2 L	$0004	Locked bit: the file is locked (read - only)
3 R	$0008	Read permission: open for users other than 0
4 W	$0010	Write permission: open to users other than 0
5 X	$0020	Execute: available to users other than 0
6 S	$0040	Symbolic link: multiple names for a file
7	$0080	File with address in directory entry
8	$0100	Hidden file, not shown by dir
9	$0200	Boring bit, don't bother to back up

Attribute bits are set using the `FILEMODE 1 mask filenames` command. The first argument (1) specifies that attributes are to be set on. The MASK is a bitmask of those attributes which are to be set; this is obtained by logically ORing the desired bits together. A list of the files to be altered follows the MASK.

Attribute bits are cleared using the `FILEMODE 0 mask filenames` command.

The BACKUP bit in a file's attribute field indicates that the file has been backed up somewhere. The DIRECTORY bit indicates that the directory entry refers to a sub-directory, not to a file. If the LOCKED bit is set the file cannot be altered (until you unlock it). For obvious reasons, FILEMODE ignores any instruction to alter the directory bit, or the file address bit, since you don't *really* want to alter them.

The read, write and execute (RWX) bits apply only to users other than user 0. User 0 can use files regardless of these three bits. However, if user 1 (etc) tries to, say, read a file without a read (R) permission, they will get an error message `Cannot open .... Permission denied.` Normally, you are always (by default) User

0, however up to 64k of user numbers can be established, for use when the Applix is serving multiple users, or running as a Bulletin Board. Normally you just ignore this.

Files with the address in the directory entry provide a way for small files to load quicker than normal.

The BORING bit is provided for you to apply to files which don't change, such as programs in the `/bin` directory, therefore removing any need to back them up. Hidden files are for tidy housekeeping when others are using the system, as User 0 will see all files in any case. Symbolic links are for use by Jeremy Fitzhardinge, and allow a file to have multiple names, provided you have an appropriate driver.

Directory related commands

Directory entries for the last 20 files or directories used are kept in a RAM cache. This increases disk performance in accesses to files that are several directory levels deep. Directory thrashing on floppies is decreased. The number of directory entries to be cached can be altered in your `mrdrivers` file.

Directory listings

DIR [pathname1] [pathname2] ....
DIRS [pathname1] [pathname2] ....

Both of these commands print out the names of some or all of the files in directories.

If none of the optional pathnames are supplied then directory listings of all of the files and directories in the current directory are presented. If some filenames are given then directory listings are given only for those files whose names appear in the filename list.

The DIRS command is a short form listing which prints out file names only; the DIR command gives a long form listing. DIRS is readable even on displays narrower than the usual 80 characters.

The command also summarises the bytes and the disk blocks used in that directory, the number of files in that directory, and the total blocks used and free on the entire device. If there is only one file, it says `1 file`, not `1 files` (unlike certain well known operating systems).

The long form directory listing of a file gives its attribute bits, the user ID number, length in bytes, load address (if any), time and date of the file, and its name.

The attributes may be ‘D’ (a directory), ‘A’ (backed up file), ‘L’ (locked file), ‘R’ (read), ‘W’ (write) or ‘X’ (eXecute). All except the ‘D’ bits may be set or cleared using the `filemode` command. The read, write and execute attributes are ignored if you are user 0 (the default). The RWX attributes are provided to assist in managing a multiple user system (say a bulletin board or networked 1616), and apply only to user IDs other than user 0. See `option .16 .56` for setting the file creation mask for the default attributes. The file creation mask defaults to .56, which sets RWX permissions on, and A and L off, when a file is created.

Set the `setdate` command for setting the time and date. See `option .17` for enabling or disabling lower case filenames.

Wildcards are very useful for obtaining partial directory listings. If for example you wished to obtain a directory listing of all the assembler source code files on your disk you would enter the command `DIR *.s`.

Entering `dir *` lists the contents of all the directories immediately below your current directory, and then lists the contents of the current directory.

Changing the current directory

CD	Display name of current directory
CD path	Change to a new directory

The CD command with no arguments causes the display of the name of the current directory. You can ensure the wole directory path appears in your prompt by using the `option 0 1` command

The CD command followed by the name of a directory causes that directory to become the current directory.

`CD /f0`	Makes the root directory of floppy 0 current
`CD ..`	Moves up a directory level
`CD mydir`	Moves down a level
`CD ../mydir`	Moves across a level

The system changes back to the current directory after a reset, from Version 4 on. Older versions did not.

Creating a directory

MKDIR path [path ...] Make directory, or multiple directories.

The MKDIR command creates a new sub-directory as specified by PATH. The directory size is rounded up to a multiple of 16 each time it runs out of space for new files. Remember that very large directories make it hard to find things, and may slow down the system. There is no provision for shrinking the size of directories if the number of files decrease.

Setting the execution search path

XPATH	Display path setting
XPATH -	Clear all paths
XPATH path1 [path2] ...	Set paths
XPATH + path1 [path2] ...	Add paths

It is often desirable for the system to search a number of directories for executable files when they are to be loaded and run. Suppose, for example, that you wish to run the 1616 assembler, SSASM. It possibly resides in /F0/BIN, or in /RD/BIN, or in /F1, etc. It is much simpler to simply type ‘SSASM’ and let the system do the searching. The XPATH command allows you to specify which directories are to be searched when the system is looking for `.xrel`, `.shell` and `.exec` files to execute.

When the name of an executable file is typed, the system searches the current directory, followed by those directories specified in the execution path, as entered by the XPATH command. If `option .19 1` is set on, the `xpath` directories (normally cached) are re-read every time an unrecognised command is encountered. Turning `option .19` off gives quicker turnround on mistyped

commands, however it means that if you add new files or swap floppies, the new files or floppy contents will not be recognised by `xpath`. You can compensate for this by doing an `xpath` with no arguments when you change floppies.

XPATH with no arguments causes the current execution path setting to be displayed. XPATH also scans all the new execution path directories. All `exec`, `xrel`, and `shell` file locations are cached in RAM, providing quicker access to executable files.

The `xpath` directories are automatically re-read after a system reset. There is a problem if you refer to a disk using a volume name; the cache is not updated.

XPATH **-** causes the current execution paths to be removed: no directory searches (except for the current directory) will occur.

XPATH path1 path2 ... instructs the system to seach the named directories for executable files. Note that the pathnames should refer to directories in which to search, not to files.

XPATH + path1 path2 ... instructs the system to add the named directories to the execution search list.

Naming disks

VOLUMES

The `volumes` command displays the name of each disk, as does a `dir`. Handy for keeping track of which disk is which, provided you remember to give the disk a name when formatting it or later.

Assigning and substituting pathnames

ASSIGN	display all current assignments
ASSIGN -	delete all current assignments
ASSIGN path1 path2	path2 is substituted whenever path1 appears

Used to replace the leading parts of a pathname. Path1 can be any convenient abbreviation. Whenever path1 appears, it will be replaced by path2, which could be some long, or inconvenient, pathname. You can define up to 20 ASSIGNs. Unlike the earlier MRD (used prior to Version 3.2c), there is no ON/OFF switch.

`assign /inc /f0/asmfiles/includedir` ensures that from then on, when a reference is made to a file or directory whose pathname starts with `/inc`, the string `/f0/asmfiles/includedir` will be substituted. Particularly handy when including files in C or assembler. Use it also to standardise pathnames of include files.

Normally `assign` expands its second argument out to a full pathname, before storing it away. However, if the second argument is preceded by a \ character, then it is taken literally. For example:

```
/f0/mdir>ASSIGN /execute ../bin
```

will assign `/execute` to `/f0/bin`, whereas
`/f0/mydir>ASSIGN /temp \.`
will assign `/temp` to `.` , which would let processes place their temporary files in their current directory.

Set an environment string

SET — Display current settings

An environment string is a method of passing information about your computer to a program, or to substitute (similar to `assign`) one term for another term or terms. It can be used to change the names you use for commands.

SET - — Deletes all current settings.

SET name1 — Deletes the current setting for the environment string with name ‘name1’.

SET name1=setting1 [name2=setting2 ...]
SET "name1=setting1 with spaces in it"

Sets the environment string with name ‘name1’ to setting ‘setting1’. If the desired setting (or name) has special characters such as space tab ! * ? | ^ > < } ; then the argument to ‘set’ should be wrapped in double quotes as shown.

SET -e name1=setting1

Sets the environment string with name ‘name1’ to setting ‘setting1’, making ‘name1’ suitable for command aliasing, so the command interpreter will repace ‘name1’ with ‘setting1’ when ‘name1’ appears as the first part of a command.

For example, `set -e del=delete` for MSDOS types. Now you can type `del`, and the `delete` command will be executed. If an environment string does not have the -e flag it can still be used as a command alias by preceding it with a ‘$’ character.

SET -a name1=setting1

Sets the environment string with name ‘name1’ to setting ‘setting1’, making ‘name1’ suitable for aliasing anywhere in the command line, so the command interpreter will repace ‘name1’ with ‘setting1’ when ‘name1’ appears ANY-WHERE in a command line.

For example, `set -a work=$home/mydir` allows you to abbreviate pathnames to `work/myfile`, (without a ‘$’ character). If an environment string does not have the `-a` flag you can still force substitution by preceding it with a ‘$’ character.

`set` is a 1616/OS inbuilt command

The `set` command permits the manipulation of the environment strings which are associated with your current interactive command shell. Each command shell can have a different environment, thus allowing multiple users to customise their own environment. An environment string consists of a name and a setting, each of which is an arbitrary length ASCII string.

The `sset` program is a front end to `set` which records your current environment string settings in the file '$home/settings.shell' in a format which is suitable for restoring your environment settings when you log in to the system at a later time. '$home/settings.shell' is normally executed for you when you log onto the system. You should generally use `sset` rather than `set`.

Environment settings can be substituted anywhere in the command line, regardless of their mode by preceding them with a '$' sign. You can further bind the substitution by parenthesising the name. For example,

Command	Output
`set fred=frog`	
`echo $fred`	`frog`
`echo $(fred)`	`frog`
`echo $fred.c`	`frog.c`
`echo /bin/$fred/xx`	`/bin/frog/xx`
`echo aa$fredbb`	`aa$fredbb  (sic)`
`echo aa$(fred)bb`	`aafrogbb`

The environment string named 'path' has special meaning for execution path searching. See the `xpath` command help for details.

All these substitutions are performed by the command line interpreter. The file system pathname parsing code also understands environment string substitution, so you can use the $(name) or $name mechanisms wherever you use a normal pathname. For example if you are within the editor and wish to write a marked block of text out to a file in your home directory you may type ^KW and enter `$home/myfile` when the editor prompts you for the output filename. The environment string substitution is done by the operating system, not the editor. The '$' character is mandatory for pathname substitution.

Environment strings may be used for communicating information to programs which you run. For example, any program which wishes to know how wide your screen is can look up the 'screenwidth' name in your environment.

Note that settings are associated with a shell type process, so if you shell escape from a program such as the editor (command line prompt contains two '>' characters) then any new settings you make will be lost after you quit from that shell. If you use `sset` under these circumstances then the desired setting will be recorded in `$home/settings.shell`, but another `sset` from a lower level shell will cause the loss of the new setting. Use `sset -l` to reload settings from `$home/settings.shell`.

Expect some more details of this command when I figure out all the neat uses of it.

Cassette tape commands

The 1616 cassette storage system is a classic example of late twentieth century technological development. As you can see from the copious attention given it (both pages), we figure most users aren't exactly heavily into cassettes, even at 3000 baud. However, if you haven't got disk drives, you can still make a very fair go at working with the Applix 1616, using either cassettes, or another computer over the serial ports, for permanent storage. Set up a large ram disk, and remember when the power goes off, so does the disk. Just don't switch off the power very often. Incidently, you can use the ***syscall*** mechanism to do raw cassette reads and writes for test purposes. Here goes:

Saving files on tape

TSAVE pathname1 [pathname2] [pathname3] ....

This command saves the named disk files on tape. Before entering this command first rewind the tape, put the player into record mode and allow the tape to move forwards until the leader is no longer over the tape head. Use the Alt T command to toggle the cassette relay during this operation. Type in this command when all is ready; the 1616/OS cassette drivers will turn the motor on and off as required.

The cassette stop/start relay must be connected for cassette I/O to work successfully.

Archiving files on tape

TARCHIVE pathname1 [pathname2] [pathname3] ....

1616/OS keeps track of those files which have been altered since they were loaded from or saved to cassette. These files appear with an 'A' in the left column of their directory listing (see the DIR command). The purpose behind this is to remind you which RAM disk files have been altered and should be backed up on cassette before you turn off your 1616.

The tape archive command saves on tape (as in TSAVE) only those files that

a) Appear on the command line as 'pathname1' or 'pathname2', etc.
   and
b) Require backing up.

Suppose for example that you are working on some assembly language programs and at the end of the session you wish to save all the files that you have newly created onto tape. A typical sequence of commands would be:

```
DELETE *.bak        ;Remove editor backup files
TARCHIVE *          ;Save all new files
```

This will automatically save new files, skipping ones that have not altered, such as the executable binary file which contains the assembler, SSASM.

Loading files from tape

TLOAD [pathname1]

The TLOAD command loads the next file from the tape into a disk file. If ‘pathname1’ is given then the tape file is written into a disk file of this name; if the optional name is omitted then the disk file has the same name as that of the tape file, which in turn has the same name as the disk file from which it was created.

Loading multiple tape files

ITLOAD

Mnemonic: Indefinite Tape LOAD

This command indefinitely performs TLOADs. Files are read off the tape until the disk to which they are being written is full, or until you reset the 1616 (using the reset switch or the Alt Ctrl R command), or until you use the Alt C command. The 1616 may take some time to respond to the Alt C command.

Verifying tape files

TVERIFY

The TVERIFY verifies tape files by reading them in the same manner as in the TLOAD command, however the actual data which is read is discarded and no disk files are created. This command allows you to verify that files that you have just saved are correctly recorded.

Memory manipulation commands

The memory manipulation commands are useful for debugging software and hardware. A safe memory area for experimenting with these commands is the $8100-$10000 memory range. Between the inbuilt monitor, the inbuilt editor, and the raw disk ***syscall***s, you can do from the keyboard nearly everything the MS-DOS version of Norton's Utilities can do, without even firing up a disk drive. I'll explain more on using these in the *Assembler Manual*, since they are a bit heavy for a beginner.

Examining memory

MDB [a1] [a2]
MDW [a1] [a2]
MDL [a1] [a2]

Mnemonic: Memory Dump (Byte, Word or Long)

These are the memory dump commands. MDB dumps memory in byte format, MDW in word format, MDL in long format. All dump formats display the data in ASCII form on the right hand side.

With no arguments the dump commands display 64 bytes of memory,starting from the address at which the previous dump finished.

If the 'a1' argument is entered then the 64 bytes of memory at this address are displayed.

If the 'a1' and 'a2' arguments are entered then all of the memory between these addresses is displayed.

Long memory dumps may be halted using Alt C.

Continuous memory examination

MRDB a1
MRDW a1
MRDL a1

Mnemonic: Memory Repetitively Display (Byte, Word, Long)

These commands continuously display the value of the byte, word or long at address a1. This function is useful for examining addresses which are being changed by interrupt routines, but it is mainly for monitoring I/O addresses during hardware debugging and development.

The Alt C terminates memory examination.

Memory alteration

MWB a1 [n1] [n2] [n3] ....
MWW a1 [n1] [n2] [n3] ....
MWL a1 [n1] [n2] [n3] ....

Mnemonic: Memory Write (Byte, Word, Long)

These commands write byte, word and long sized numbers into memory starting at address ‘a1’.

If any of the numbers n1, n2, etc. are entered then this data is sequentially written into memory starting at a1, with the correct data size. No read is performed upon the addresses which are written to.

If none of the optional numbers are supplied then the user enters a prompted input mode. The current address and the value of the byte,word or long at that address is printed out and you may do one of the following things:

1. Type in a new value to be written at this address
2. Press Enter to go on to the next address
3. Type ‘**r**’ followed by Enter to go back to the previous address
4. Type ‘**.**’ followed by Enter to quit from prompted input mode

Putting ASCII strings in memory

MWA a1 [string]
MWAZ a1 [string]

Mnemonic: Memory Write Ascii (Zero)

These commands write an ASCII string into memory starting at address ‘a1’. The MWAZ command appends a null (zero byte) to the end of the string. If the optional string is supplied then the string is simply written into memory.

If the string is omitted then the user enters a prompted mode of input where the current address is displayed and strings are typed in. In this mode the following things may be done:

1. Type in a new string to go at this address
2. Press Enter to go on to the next address
3. Type ‘**r**’ followed by Enter to go back to the previous address
4. Type ‘**.**’ followed by Enter to quit from prompted ASCII input mode

In both forms of these commands only a single string may be entered at a time. If your strings have spaces or tabs they must be surrounded by double quotes.

For example, the command

```
MWAZ 8000 "Test string "
```

writes that string, null terminated, into memory at address $8000.

Memory filling

MFB a1 a2 n1
MFW a1 a2 n1
MFL a1 a2 n1
MFA a1 a2 string

Mnemonic: Memory Fill (Byte, Word, Long, Ascii)

These commands fill the memory address range ‘a1’ through to ‘a2’inclusive with the supplied value. The byte, word and long size commands interpret ‘n1’ as an appropriately sized number and write it into memory throughout the range in address increments of one, two or four bytes.

The MFA command fills the memory address range with the supplied string. The string must be surrounded by double quotes if it contains spaces or tabs. The string is not null terminated. If the length of the address range is not an even multiple of the length of the string then the extra characters are written to addresses beyond ‘a2’.

Memory comparing

MCMP a1 a2 a3

Mnemonic: Memory CoMPare

This command compares two blocks of memory bytes. The memory block between addresses ‘a1’ through to ‘a2’ is compared to the block starting at address ‘a3’. Any differing bytes in the two blocks are printed out in the form

address1-b1 address2-b2

where ‘address1’ is the address within the a1-a2 block where the difference was found, ‘address2’ is the address within the a3 block, ‘b1’ is the byte at address ‘address1’ and ‘b2’ is the byte at address ‘address2’.

Large amounts of output from the MCMP command may be interrupted with the Alt C key sequence.

Memory searching

MSEARCH a1 a2 n1 [n2] [n3] ....

The memory search command searches the given address range for the specified byte pattern. The byte pattern must be at least one byte long - specifying the optional arguments ‘n2’, etc. allows you to search for multi-byte patterns.

The start address of any found pattern is printed out. Use Alt C to stop large amounts of output.

Example:

```
MSEARCH 500000 50ffff 41 70 70 6c 69 78
```

This command searches the 1616 ROM space for the sequence of characters in the word ‘Applix’. The ASCII codes for the characters in this word have been specified.

Memory moving

MMOVE a1 a2 a3

The MMOVE command moves the contents of the block of memory in the address range ‘a1’ through to ‘a2’ inclusive to the address ‘a3’ and onwards.

There are no restrictions on the move: overlapping moves are handled correctly.

Saving memory in a file

MSAVE a1 a2 pathname1

The MSAVE command saves the contents of the address range ‘a1’through to ‘a2’ inclusive in the named file.

Loading memory from a file

MLOAD pathname1 [a1]

This command loads all the bytes in the named file directly into memory at the given address (if supplied). If the address is not given then the file’s contents are loaded at the address specified in the load address field of the file’s directory entry. An error message arises if this address is not a reasonable one.

Moving characters about

CIO [n1]

Mnemonic: Copy Input to Output

The CIO function is a general purpose 1616 function for moving characters about between devices and files, between devices and devices and between files and files.

This command simply reads characters from standard input and writes them to standard output, optionally terminating on the character whose ASCII code is ‘n1’. Termination also occurs when the Alt Ctrl C key sequence is entered, or when an end-of-file character (as specified by `option 6`) is read from a character device.

If the terminating byte is specified it must appear on the command line before any I/O redirections.

The power of this command comes from 1616/OS’s I/O redirection capabilities. The input and/or output of the CIO command is redirected to different devices to move character streams about.

For performing file transfers from other machines you may specify the optional termination character ‘n1’. If, for example, you were downloading a file from a CP/M machine to the 1616 you may select the CP/M end-of-file character (1a hex) as the terminating character. CIO copies all characters up to, but not including the termination character. Use `option 6` to set the end of file character.

When CIO gets its input from a file (using the ‘<filename’ redirection) it ends the transfer when the file is exhausted. This is the program that underlies the `cat` and `type` commands.

Some examples:

`CIO <filename`

With this command we have left the output device at its default, the video display. The input for the CIO command comes from the file ‘filename’ and the command will terminate when all characters have been read from this file. This command prints the file on the screen!

`CIO 1a <sa: >myfile`

Read characters from serial channel A and write them onto the new file ‘myfile’. Terminate when a control-Z (ASCII code $1a) is received.

`CIO >>myfile <sb:`

Read characters from serial channel B and append then to ‘myfile’. The user will have to type Alt C to terminate this transfer.

`CIO <sa: >sb:`

Copy bytes from serial channel A onto serial channel B until Alt C is pressed or an end-of-file (normally control-D) is received.

`CIO <myfile >cent:`

Print the file ‘myfile’ out on the parallel printer.

Shell file commands

Echo command line arguments

ECHO [-n] [arg1] [arg2]

The ECHO command simply prints its arguments out. If the first argument is ‘-n’ then a new-line is not printed out after the arguments. This command is mainly used within `shell` (executable text) files.

For example, suppose one creates a `shell` program file (called ‘sum’) containing the following lines:

```
ECHO -n "Adding " $1 " and " $2 " produces " EXPR $1 + $2
```

When this `shell` program is executed with two numeric arguments (e.g. ‘sum 1 2’) the appropriate output results.

You can echo characters that have an audible effect, such as the Ctrl G or Ctrl B beeps. Note that you can alter the volume and sound of a beep using the oscontrol() system call.

Pausing

PAUSE n1

The PAUSE command causes 1616/OS to cease processing commands for a duration of ‘n1’ twenty millisecond system timer ticks. In otherwords the pause length is (n1 / 50) seconds long. An Alt C interrupts a pause.

PAUSE is designed for inclusion in `shell` program files to allow users time to read the contents of the screen before new commands from within the `shell` file are executed.

Remember that the default form of numeric input is hexadecimal! Put a full stop (‘**.**’) in front of decimal pause lengths.

For example consider the following `shell` program (called ‘memex’):

```
ECHO "To observe memory alteration"
ECHO
ECHO "Memory before executing " $3
MDL $1 $2
PAUSE .100              ;2 seconds
```

```
$3 $4 $5 $6
ECHO "Memory after executing " $3
MDL $1 $2
```

We may use this program to observe how a range of memory changes before and after a transient or inbuilt command is executed. For example the command

```
MEMEX 10000 12000 mfb 10000 12000 12
```

will cause the following:

1) The messages and memory contents are printed out
2) The system pauses for 2 seconds, allowing time to observe the output, or to type Alt S for more time.
3) The command `mfb 10000 12000 12` is executed.
4) The memory range is printed out again.

Setting the time and date

SETDATE year month day hour minute second

SETDATE is used to inform the system of the current time and date. When the time and date is correctly entered with this command it is echoed in the standard format.

All six numeric arguments must be given. The format is:

year	0 - 99
month	1 - 12
day	1 - 31
hour	0 - 23
minute	0 - 59
second	0 - 59

Remember to include the leading '**.**' when specifying a decimal number.

Example:

```
SETDATE .92 4 1 .12 .30 0
```
is half past twelve on April fools day.

Disk users can buy a $50 package of plug in real time battery backed SmartWatch, and routines for setting and forgetting it, and showing the time in a corner of the display.

Users of 1616/OS V4.2e may note that on a certain date of the year, a special message will appear on reset; this indicates Andrew was programming eproms on his birthday. Incidently, I haven't a clue where in the eproms he hid the text of the message, but look for his name with high bit set.

Displaying the current time/date

DATE

This command causes 1616/OS to print out the current system time and date. If the time and date have not been set up then the output will contain many zeroes and the month field will appear as question marks.

Executing machine code

GO a1

The GO command causes 1616/OS to perform an MC68000 'JSR' instruction to the program at address 'a1' (presumably the start address of a machine language program), and sets up an argument array.

A program which is executed in this manner must preserve the MC68000 stack frame and return to 1616/OS with an 'RTS' instruction. The value which the program returns in the MC68000's d0 (data register zero) is printed out by 1616/OS when it regains control.

Manually performing system calls

SYSCALL callno n1 n2 n3 ....

This command causes 1616/OS to perform system call number 'callno', with parameters n1, n2, etc. The value returned from the system call (in data register d0) is printed out upon return from the system call.

If a non-numeric argument is given then the system call is passed a pointer to the non-numeric string, rather than an evaluated number, as with numeric input, so the following works:

```
SYSCALL .48 "%d plus %d = %d" 2 2 4
;printf system call
```

Refer to the 1616/OS *Programmers Reference Manual* for more (many) details about system calls.

Process status

PS

Displays a list of all current processes from the process table.

Show their PID (process ID), and their parent's ID (PPID), the amount of time they have consumed, their current status, their load address, stack area, their current stack pointer value, the program counter value when they were last descheduled, and the handles for their standard input, output and error streams. Finally, their name is given.

PID	the process's ID number, in decimal.
PPID	the process's parent process PID.
HPID	home PID
TIME	the number of 20 millsecond time slices over which the process has run, converted to minutes and seconds.
Status	This represents the state of five flags in the process's process table entry. They are as follows:
W	Set if the process is waiting (blocked) on another.
S	Synchronous.
A	Asynchronous.
E	Exit pending: process about to depart.

B	Binary: process memory to be freed on exit.
K	Killed: some one has killed this process.
Load	The load address of the program; actually the call address passed to the ***schedprep*** system call when the process was scheduled.
Stack	The address of the process's stack area.
SP	The current value of the process's stack pointer.
PC	The value of the process's program counter when it was last descheduled.
I, O, E	The handles of the process's standard input, standard output and standard error streams.

The last field is the name of the process. This name may be used when referring to the process, for example, using `kill`.

See `kill` for getting rid of unwanted processes. Careful readers may note that Andrew tends to change the exact information given in each release of the operating system. If you notice a change and want it explained, just contact me and I'll add it next release.

Kill a background process

KILL process-name
KILL [-ak*nn*] processID

You can stop a process by name or by PID number. The `ps` command will list both name and PID number of all processes. Killing by name is usually easier, however if you have several processes with the same name, using the PID number ensures the correct one is killed. Kill emits a "PID: terminated" message via the appropriate output device. The following options are available:

-a	kill all children processes
-k	unconditionally kill
-*nn*	specify which signal to send, in decimal, rather than the default sigterm which is 15. See the *Programmer's Manual* for details of signals available, which generally follow UNIX conventions.

Wait for a process to end

"WAIT command ! action" &

The `wait` command returns when the selected PID number has terminated. It may be used to synchronise an asynchronous process, by letting you (or another program) know when it has ended.

For example, suppose there is a background process called `download` running. We wish to know when this has completed. The command to do this is:

`"wait download ! echo` Ctrl G Ctrl G Ctrl G `> con: " &`

This command will suspend until the completion of `download`, and then emit some beeps.

Pipes

command | command

A pipe is created by typing vertical bars (|) between two or more commands. The operating system starts each command asynchronously. An interprocess pipe connects the output of each command with the input of the next. If the last command in the pipeline was run synchronously, the system blocks until it is complete.

You can also pipe standard error using the **^** symbol, however this can not be done while also using the normal pipe.

Under multitasking, all the specified tasks are run simultaneously, rather than sequentially via temporary files, as is the case for the older MRD, or for MS-DOS.

Altering internal settings

OPTION optionnum setting

The option command is a general way of varying various fiddly settings within 1616/OS. If you use the option command without a second (setting) parameter, it will return the current setting of the option. This is handy for finding things like the present end-of-file character.

The present options are:

option 0 1	(default) Turns on display of the current directory in the prompt. option 0 0 turns it off again.
option 1 1	Turns on verbose mode. Many commands operate quietly. option 1 0 turns verbose mode off.
option 2 2	(default) Turns on alphabetic sorting. The option 2 setting also affects the sorting of wildcard expansion. option 2 0 turns off sorting of directory listings. option 2 1 turns on sorting of listings by date.
option 3 0	Affects how information is displayed when a machine exception occurs. If option 3 0 (default) has been selected then the screen is not cleared and a register dump only is displayed. The system registers are dumped in memory as follows: $8000 registers d0-d7/a0-a7

$8040	access address
$8044	function code
$8048	instruction register
$804c	status register
$8050	program counter
$8052	pointer to a string describing the exeption type.

If bit 0 is set (3 1) the screen is cleared, the registers are dumped and a stack backtrace is displayed.
If bit 1 is set (3 2), exceptions cause the offending process to be killed, rather than rebooting the machine. This can be handy if you are doing something that may crash the system, while important things (like a huge download) are happening in background.
If bit 2 is set (3 4), halt system until reset upon an exeption. If bit 3 is also set (3 c or 3 d), the first exeption only is recorded in memory at $8000 on.

option 4 1 — (default) enables the printing of error messages from the floppy disk driver code.
If option 4 0 is in effect these messages do not come out.

option 5 0 — (default) getmem returns a negative error code when out of memory.
Option 5 1 causes the system to generate an internal error when the memory allocation function getmem receives a request for more memory than is available.

option 6 N — Sets the end-of-file character for character devices. The normal setting is $100 (no EOF character). Reads from character devices terminate when this character is read. Setting N to a number greater than 255 effectively disables end-of-file checking. Option 6 with no parameters prints out the current EOF character (applies from Version 3.2c on). Alt Del toggles between $100 and $04 (Ctrl D as EOF).

option 7 8192 — Sets the `exec` file stack space. You can tune it to suit the needs of your program.

option 8 0 — (default) Disables the system from writing to the system blocks of a disk (blocks zero through to the start of the root directory). This is a safety feature in the blkwrite system call which will hopefully prevent the odd disk crash.
option 8 1 enables writing to the system tracks. Turn this option off after use.

option 9 1 — (default) Enable the Alt Ctrl R keyboard reset function.
option 9 0 disables.

option .10 1 — (default) Enable the Alt S output suspension facility. option .10 0 disables. Note the period (.) indicating a decimal value,

option .11 1 — (default) Enable all special keyboard Alt codes. option .11 0 disables.

option .12 1 — (default) Enables the output of a beep character when the system prints out an error message. option .12 0 disables.

option .13 1 — (default) Warm start upon bus or similar exception. option .13 0 is cold start (available from Version 3.1e on, previous default was cold start)

option .14 1 — (default) Dumps contents of program counter and registers, if the 68000 trace flag is set on. You set the CPU trace flag in the 68000 status register with `or.w #$8000.sr`, and clear it using `and.w #$7ffff.sr`. option .14 0 disables PC and register output.

option .15 0 — If bit 0 (.15 1) is set, do not create `.bak` files in `edit` and `Dr Doc`.
Bit 1 (Alt C interrupt ... broken)
If bit 2 is set (.15 4), the 5 key on the numeric keypad generates Ctrl Q in non-numeric mode.

option .16 .56 — Set file creation mode, the default `filemode` mask. You are setting a bit pattern here. The normal default ensures all files have RWX permissions for all users. This can be altered for greater security when there are multiple users. Probably not available this time round.

option .17 1 — Enable lower case pathnames, in directories.

option .18 0 — Set output video mode.
If bit 0 is set (1), all escape code sequences are ignored. Characters are simply printed out. Remember, we are setting bits, not giving a value.
If bit 1 is set (2), all control characters remove their special meaning, **even** carriage return, line feed, etc.
Obviously, if you set both bits 0 and 1 (3), the display goes into a monitor mode, and displays all characters sent to it (showing their ASCII equivalent where required).
A ^L clears the screen if bit 2 is clear, and tabs overwrite characters.
If bit 2 is set (4), the terminal emulation is closer to Televideo 950 standard. ^L moves cursor forward, while tabs are non-destructive, and move cursor to the next tab stop.

option .19 1	Enable automatic re-read of xpath directories, in the event of a filename error. Bit 0 set enables re-reading of execution paths on a command miss. Bit 1 set causes the system to scan the xpaths before reading the current directory, which is useful for hard disk users.

Note that if you call an option with -1 (or error) as the second parameter, the next option called will return a bad option error message. If the second option given is correct, it will be set correctly, despite the error message returned.

Quitting the command interpreter

QUIT

The QUIT command is provided as a way of returning to an application program which temporarily called the 1616/OS command interpreter. For an example see the documentation for the 1616/OS editor Ctrl K I miscellaneous command.

You can also use the end-of-file character (usually Ctrl D) to quit, provided you have changed the value of the EOF character (normally not implemented) to Ctrl D using `option 6 4` or the Alt Del hotkey.

Specifically, this is the way for the user to terminate a program's call to the ***iexec*** system call.

Handy utilities

Numeric base conversion

BASE n1 [n2] [n3] ....

The BASE command converts the supplied numbers into their binary, decimal, hexadecimal and octal equivalents. The output is now in neat columns, to please Andrew McNamara.

Entering the editor

EDIT filename1 [n1]

This command invokes the 1616/OS full screen editor. The filename is the file which is to be edited; if it does not exist it is created when the editor writes its text out. The optional numeric argument, 'n1' is the tab stop width which the editor is to use (default is 8). This may not be an inbuilt command in versions of 1616/OS V3.0 running in 256k EPROMS. See the chapter on editing for the full command list. Most commands are WordStar compatible.

Assembling 68000 code

SSASM filename

This command invokes the 68000 assembler, which is in 512k EPROM from version 3.2c to Version 4.1. This is not included in EPROM from version 4.2a on, but is provided on disk. See the *Assembler Manual* for details of usage and options.

Expression evaluation

EXPR n1 [op] [n2] [op] [n3] ....

This is a simple expression evaluator. You type in numbers separated by arithmetic and logical operators and the result is calculated and printed out. All the operators and numbers must be separated by at least one space or tab.

All of the calculations are done with 32 bit integers. This means you can't have fractional numbers, or numbers that include a decimal point. The results are displayed in binary, decimal and hexadecimal. It does not do floating point arithmetic!

The evaluator ignores operator precedence: expressions are simply evaluated from left to right. Parentheses are not understood.

If your first attempt at a calculation fails, remember last line recall!

The available operators and their meanings are:

x or X	multiplication
/	division
+	addition
-	subtraction
%	modulo (remainder)
&	bitwise AND
|	bitwise OR
^	bitwise exclusive OR

The ‘X’ is used for multiplication because the ‘*’ character is expanded out into the names of all the files on your disk before the command line is processed! You can however use the * symbol, provided you escape it with "*".

Examples:

```
EXPR .100 x .200          ;Calculate decimal 100*200
EXPR %1101 | %10000       ;Set a bit
EXPR .12345 / .63         ;How many 63’s in 12345?
EXPR .12345 % .63         ;With how many remainder?
EXPR .195 x .63 + .60     ;Check it.
```

Printing the ASCII character set

ASCII [d | h | D | H]

This command produces a printout of a table of 128 ASCII characters. The command ‘ASCII h’ results in a hexadecimal-ASCII table, while the ‘ASCII d’ command gives a decimal ASCII table of the first 128 characters. Using upper case parameters, as in ‘ASCII H’, or ‘ASCII D’, produces a table of the second 128 (mostly graphics) characters.

Timing a command

TIME command

Putting the command ‘TIME’ at the start of any 1616/OS inbuilt or transient command causes the command to be executed in the normal manner, after which the elapsed time is displayed in minutes and seconds.

For example,

```
TIME ssasm /f0/srce/myfile
```

causes /f0/srce/myfile.s to be assembled, after which the assembly time is displayed.

Defining function keys

FKEY n1 string1

This command programs function key ‘n1’ to produce ‘string1’ when typed. The ‘n1’ argument must be in the range 1 to .10 (notice that .10 is decimal, or you can use the hexadecimal equivalent ‘a’). The string of characters must be a single argument: if it contains any spaces or tabs it must be surrounded by quotes. Strings may contain any characters, including control characters (entered by preceding them with Ctrl P). Function keys may be programmed to produce more than one command line by including the **Enter** (or Ctrl M) character in the string (don’t forget to precede control characters with Ctrl P).

Remember that you can also program the function keys direct from the keyboard using the Alt Ctrl function key combination.

Ten Function key definitions of up to 63 characters in length are permitted. Examples:

```
FKEY 1 dir
```

```
FKEY 4 "Ctrl K X Ctrl P Ctrl M delete myprog.bak Ctrl P Ctrl M SSASM myprog.s"
```

The last example is a complicated but useful one. It does the following things:

1) Quits from the editor using the ^KX command (however a ^P is not required, because the editor insert mode will directly accept control characters as commands)
2) Issues an Enter (^M) character to confirm the file name (a ^P is required, to embed the Enter, because you are now outside the editor’s insert mode)
3) Deletes the editor’s backup file (again, a ^P is required, since you are on the regular Applix command line)
4) Assembles the file upon which you are presumably working

Setting video frequencies

VMODE hfreq vfreq [hoffset] [voffset]

This is provided for those running USA video monitors, and enables you to tweak the 6545 CRTC registers to produce a stable picture on your particular display. Don’t fiddle until you know what you are doing with it - read the *Hardware and Construction Manual*.

Helpful reminders

HELP [cmdname] [cmdname] ....

The command ‘HELP’ with no arguments causes a sorted list of all of 1616/OS’s inbuilt commands to be printed out.

The HELP command may be used to obtain more detailed help concerning one or more particular inbuilt commands by using the name of the inbuilt command as an argument to HELP. For example,

```
HELP mdl filemode
```

When this form of the HELP command prints out these usage messages the following conventions apply:

number	Compulsory numeric input
string	Compulsory input (often a filename)
[number]	Optional numeric input
[string]	Optional input

These usage messages tell you what input formats are acceptable to 1616/OS's command parser. Typing in commands which comply with this format will not cause a 'syntax error' message, however they may still be rejected by 1616/OS. More detailed `help` messages are given from Version 3.2c on, and this includes a full list of `option` parameters.

Reprogramming the serial ports

SERIAL channel
SERIAL channel baudrate rxbits txbits parity stopbits

This command reprograms one of the 1616's serial channel's operation modes. If only the channel name 'A' or 'B' is used, it displays the current setting of the channel in question. The arguments are used as follows:

channel — This must be the letter 'A' or the letter 'B', depending upon which serial channel you are reprogramming.

baudrate — This is the desired transmit and receive baud rate. It may be any of the standard rates, up to 38,400 baud. You may specify non-standard baud rates if you wish, however 38,400 is the suggested maximum, and baud rate accuracy will suffer if you specify non-standard rates which are above 2400 baud. Remember to add the leading '**.**' if specifying this in decimal! Actually, much faster baud rates can be obtained, however above about 57k, there is a problem entering the correct rates.

rxbits — This number specifies the number of received bits. It may be 5, 6, 7 or 8.

txbits — This number specifies the number of transmitted bits. It may be 5, 6, 7 or 8.

parity — This number sets the transmit and receive parity. 0 sets no parity, 1 sets odd parity, 2 sets even parity.

stopbits — This sets the number of transmitted stop bits. Specifying a '0' gives 1 stop bit, a '1' gives 1.5 stop bits and a '2' gives 2 stop bits.

Example:

```
serial b .4800 7 7 2 0
```

Programs serial channel B for 4800 baud input and output, 7 receive data bits, 7 transmit data bits, even parity and one stop bit.

If running 7 data bits and parity, there was a problem prior to V3.2. The parity bit is not stripped by either the SCC or by the TERMA or TERMB routines in some earlier version of 1616/OS. This may lead to strange graphics characters being included in your transferred material.

The serial receive routine masks incoming 7 bit characters by ANDing them with $7F, 6 bit characters with $3F, and 5 bit characters with $1F, to keep Andrew McNamara happy.

From Version 4, the channel settings are stored by the serial driver, so that they can be displayed upon request. The serial ports are also restored to the last settings made after a reset, rather than being initialised to the default settings. Drivers support up to four SCC serial chips.

Using the 1616 as a terminal

TERMA [-L]
TERMB [-L]
TERM SA: [-L]
TERM SB: [-L]

These commands allow you to use your 1616 as a remote terminal to another computer. TERMA uses serial channel A, TERMB uses channel B. The optional `-l` parameter copies all the material received to standard error (in addition to standard output), so that it can be redirected to a file to give you a log of a terminal session.

Characters are obtained from standard input (normally the keyboard) and are transmitted over the serial link. Characters which are sent to the 1616 are displayed on standard output (probably the video display), and optionally sent to standard error. e.g. `termb -l }} logfile`

If full cursor control is needed then the 1616's terminal control escape commands will have to be installed on the remote system. These are set out below.

You may leave terminal mode by typing Alt Ctrl C.

Version 3.0 of 1616/OS sees a change from hardware video scrolling to software scroll. Whilst this makes life easier for video programming it does slow down scrolling, so the 1616 may experience overrun problems with characters which come after a series of newline characters when running at high baud rates.

Escape sequences

If you embed terminal control characters or escape sequences in a text, you can display **bold**, underline, *italics*, $_{\text{subscript}}$, and $^{\text{superscript}}$ text (or any reasonable mixture). The $29.95 *Dr Doc* editor uses these, as does anything that writes text to the display. These sequences are displayed or used whenever the Applix 1616 encounters them, thus making it relatively easy to employ fancy text in your programs. Test them by using Ctrl P to embed an Esc key in your text. Thus, from the keyboard, `echo` Ctrl P Esc G 4 will put the display into **bold** mode, etc.

^G	Beep speaker 7.
^I	Tab 8.
^J	Line feed 10.
^K	Cursor up 11.
^L	Clear screen 12.
^M	Carriage return 13.
^V	Cursor down 22.

^^	Home cursor 30.
ESC =	(row+32) (col+32) Positions the cursor.
ESC)	Start highlighting.
ESC (	End highlighting.
ESC *	Clear the screen, or current window.
ESC B	(value+32) Sets the background colour to ‘value’.
ESC b	Visible bell.
ESC E	Insert a line at the current one.
ESC F	(value+32) Sets the foreground colour to ‘value’.
ESC G 1	Sets subscript mode.
ESC G 2	Sets superscript mode.
ESC G 4	Sets bold mode.
ESC G 8	Sets underline mode.
ESC G @	Sets italic mode.
ESC G 0	Clears subscript, superscript, underline, bold and italic modes.
ESC I	Back tab.
ESC j	Reverse scroll display.
ESC M	(from+32) (to+32) Copies the contents of a line.
ESC P	(position+32) (value+32) Writes ‘value’ into the 1616 video pallette at ‘position’.
ESC Q	Character insert.
ESC q	Enter insert mode.
ESC r	End insert mode. ($1B, $72).
ESC R	Delete the current line.
ESC S	(value+32) Sets the border colour to ‘value’.
ESC t	Clears from cursor to the end of the line.
ESC T	Clears from cursor to the end of the line.
ESC W	Delete character.
ESC Y	Clears from cursor to the end of the screen.
ESC . 0	Cursor off
ESC . 1	flashing cursor
ESC . 2	steady block cursor
ESC . 3	flashing underline
ESC . 4	steady underline
ESC 0x03 nn	Delay video by nn ticks

Downloading S-records:

SREC [filename] <redirection

The Motorola S-record downloader in 1616/OS is available for transferring to the 1616 binary executable code which has been compiled or assembled on another computer system.

S-records are an ASCII representation of executable machine code (see Appendix B).

If the filename is given the binary code is transferred to a file of that name. If the name is omitted then the code is loaded directly into the 1616’s memory.

You must use input redirection (with the ‘<’ symbol) to specify the device from which S-records are to be loaded. By obtaining input from a file one can convert S-record files into binary ones within the 1616.

The SREC command simply prints out any characters which are received from the input until a start record (S1, S2 or S3) is received. Bytes are loaded in and the checksum is verified. Loading terminates on an S7, S8 or S9 record.

Example:

```
srec myfile <sa:
```

loads S-records from serial channel A, putting the binary code into ‘myfile’.

9
Appendix A

For MS-DOS users

This short appendix is intended for readers who are already familiar with one or more computer systems. If you are familiar with some other computer system, you may prefer to leap in, without looking through the manual. Experienced MS-DOS and UNIX users should have little difficulty, although many commands are different. Please remember however that you will not be able to make fully effective use of this system until you know the facilities available. This manual contains a complete list of all commands, with their syntax, and examples.

Common commands

help	Lists all commands. Followed by command names, it tells how to use those commands.
dirs	Short list of files and directories.
copy	Copies source to destination, for files and devices.
type	View contents of a file.
cd	Change from one disk or directory to another.
mkdir	Make a new directory.
delete	Delete a file, or multiple files, or empty directories.
xpath	Tells which directories to search for programs.
rename	Changes the name of a file within a directory.
move	Changes the name of a file within a disk.
cat	Copies multiple files and device inputs to standard output.
m...	Monitor commands, for debugging, and manipulating memory.

Command line

1616/OS uses a command line, similar to MS-DOS and UNIX, however floppy drives are named **/f0**, **/f1**, hard drives are **/h0**, **/h1**. You always have a ram drive **/rd**, whose size is usually set by the **mrdrivers** file in your boot disk.

Disks, directories and subdirectories use the slash / as a divider (like UNIX, unlike MS-DOS slosh).

Input and output redirection is available using <, >, while >> appends to an existing file. The standard error output can also be redirected using **}**, and **}}**.

Extensive line editing facilities are available from the command line, together with a history command that lets you re-edit the last 10 commands. Multiple commands can appear on a line, which can be as long as 511 characters. See Chapter 2 for a list.

A 'WordStar'-like full screen editor is always available, and can be used from within languages such as BASIC. See Chapter 6 for details. There is a built-in calculator, `expr`, terminal emulation, and function key re-definition. There is a inbuilt 68000 assembler available in EPROM in some models.

Wildcards are similar to MS-DOS, however they are more consistent in use, and are available from **all** commands. For example, `type *.doc` will display the contents of all 'doc' files, while `dir *` will display the contents of all directories one level up the tree from your present directory.

A **&** after a command will invoke multitasking, and run the command in background, while you continue working. See Chapter 7 for details.

Edit quick reference

Cursor movement

^E	Up one line	^X	Down one line
^QE	Up to top of page	^QX	Down to bottom of page
^R	Up about one page	^C	Down about one page
^QR	To start of file	^QC	To end of file
^S	Left one character	^D	Right one character
^A	Left one word	^F	Right one word
^QS	Left 80 characters	^QD	Right 80 characters
^B	Start/end of line	^J	Start of next line

Scrolling

^Z	Scroll up	^W	Scroll down

Text deleting

^H,BS	Delete char backward	^G, Del	Delete char under cursor
^Y	Delete line	^T	Delete word forward
^V	Delete line backwards	^QY	Delete line forward

Block commands

^KB	Mark block start	^QB	Go to block start
^KK	Mark block end	^QK	Go to block end
^KY	Delete marked block	^KV	Move marked block to cursor
^KH	Hide block	^KC	Copy marked block to cursor
^KW	Write block to file	^KP	Block to Undo buffer

File commands

^KR	Read in (merge) a file	^KD	Write out file, continue editing
^KX	Write out file, quit		

Miscellaneous

^KQ	Exit without saving file	^KI	Escape to 1616/OS
^KE	Execute 1616/OS command	^QG	Go to line number
^K0-9	Set block markers 0 to 9	^KF	Partial screen freeze
^Q0-9	Go to a block marker	^QF	Find pattern
^QA	Substitute pattern	^L	Repeat last pattern find
^QU0-9	Review undo buffer 0 to 9	Esc	Redraw the display
^N	Repeat last substitution	^U0	Undo buffer 0 to 9
^UU	Undo most recent		

Motorola S-records

Motorola S-records are a way of representing binary memory images in an ASCII form, which is a more convenient format for transfer between some computer systems.

Binary data is transferred by sending fields of the following form:

Field	S	Type	Length	Address	Data	Checksum
Number of Characters	1	1	2	4,6 or 8	varies	2

The fields are used as follows:

S — The letter ‘S’ (upper case) signals the start of the record

type — A digit between 0 and 9 which defines the record type, as described below:

0: A header record (ignored by 1616/OS)
1: A data record with a 2-byte address field
2 A data record with a 3-byte address field
3: A data record with a 4-byte address field
4: Not allowed
5: The number of type 1, 2 and 3 records in a group of S-records. Ignored by 1616/OS
6: Not allowed
7: Terminating record for S1 records
8: Terminating record for S2 records
9: Terminating record for S3 records

length — The number of character pairs in the address, data and checksum fields. This field is a two character quantity (a hexadecimal byte)

address — This is the address at which the S-record loads into memory in the target system. It is a 2, 3 or 4 byte address (4, 6 or 8 hex characters), depending upon whether it is part of an S1, an S2 or an S3 type record.

data — This is the actual data to be loaded. It is a series of hexadecimal bytes.

checksum — This is the checksum of the length, address and the data fields. It is calculated by adding together the values of all the bytes received in these fields, inverting (taking the one’s complement of) the result and transmitting the least significant byte of the result as two ASCII hexadecimal digits.

Miscellaneous

This section explains various miscellaneous items that will not be of interest to you in your first encounter with the Applix 1616, and that do not conveniently fit into the flow of previous discussions.

Hardware exceptions

If something goes seriously wrong with a 68000 program, the microprocessor can be driven into a fault condition (called an exception) which is handled by special software. In most computers, an attempt is made to recover from minor errors, however this often (usually?) causes problems in the program in later use. In the 1616, the emphasis is placed upon ensuring that information about the exception is made available to the programmer, rather than trying to recover from the exception. Programmers generally find this helpful; as a user, you may not.

When an exception is detected, the exception type and a full register dump are displayed, whereupon the 1616 grinds to a halt. You must press reset to regain control. The `option` command may be used to vary the form of output when an exception occurs, making it possible to obtain a stack backtrace. Once you learn 68000 assembler, you will find all this of great use. See `option 3` and `option 13`.

Using reset

If the 1616 is doing something and you wish to regain control of the system, first try the Alt Ctrl C key sequence. The running program should detect your interrupt, and should stop whatever it is doing, awaiting further input.

If this fails the next approach is to try the Alt Ctrl R sequence. At a software level, this is equivalent to pressing the 1616's reset switch (at the rear). If this fails to regain control, then the 1616 is very hung up and you must press the reset switch.

Using the reset switch, or the Alt Ctrl R, sequence causes a 'level 2' reset. This regains 1616/OS control, but performs some reinitialisation of data areas within 1616/OS. In short, a reset tries to restore things to what they were before they got out of control. The contents of your RAM disk usually remain intact through this reset.

If some critical areas of memory have been destroyed by a runaway program, then a 'level 1' reset is necessary. The level 1 reset is invoked by performing two resets (by either method) within three seconds of each other. This fixes most major problems, by initialising many extra data areas..

The ‘level 0’ reset usually occurs only at power-on time. It may occasionally occur when you reset, if a program has gone berserk and overwritten reserved memory locations. A level 0 reset reinitialises everything. This includes the RAM disk, the contents of which will be lost, and the character set, including any specially changed display characters you have produced.

Switch selection - colour, boot, etc

Prior to Version 3.1e EPROMS, switch 2 (numbering 0, 1, 2, 3) of the quad DIL switch on the motherboard was used to inform the 1616/OS about the type of video monitor you are using, so that the appropriate colours or grey scales are used for your display. If you have EPROMS prior to Version 3.1e, set switch 2 *off* if using a *colour* monitor. Set switch 2 *on* if using a *monochrome* monitor.

This monitor selection is not required on Version 3.1e and later, and switch 2 is now used to select an external serial terminal as the console, for test purposes. If the switch is open (off), serial port A is used as the console, provided no disk controller is present. If you do accidently leave the switch open, a rude message appears on the video to remind you of what has happened. If you have a disk system, the selection of a serial port can be made by software on the disk, so the setting of this switch is used differently.

If a disk controller is present, switch 2 now determines whether you will attempt to boot first from the floppy drive or hard disk.

Monitor types

The best monitor for the Applix 1616 is a colour Multisync (or greyscale multisync if cost is a problem). Since these are costly, the alternatives are discussed below.

A standard Applix 1616 is compatible with an ordinary, old fashioned, IBM style RGBI CGA colour monitor. The style of monitor was selected because it is readily available, and more moderately priced than multisyncs. These displays are relatively cheap because they use standard TV video refresh frequencies, are easy to make, and are built in great numbers. It does however have the great disadvantage that, like a TV, it does not provide a very clear or readable text display. Look for them at sales! Don’t buy a new one these days, as they are obsolete.

The standard Applix can also use a composite monochrome display to display shades of grey (well, green or amber usually). You can connect a non-IBM composite video monitor as used by Apple and MicroBee (that is, you don’t need an IBM Monochrome or Hercules style monitor ... but read on). These composite monitors are considerably cheaper than colour displays. Again, check out sales, and look for prices well under $100.

If you decide upon a monochrome monitor, I strongly recommend that a ‘dual scan’ monitor be used. These are intended for use with IBM PC clones, and can readily be adjusted to accept both standard IBM RGBI CGA video (as provided by the Applix), and also the higher scan frequency IBM monochrome or Hercules graphics monochrome output. You should note that many (most) monochrome

monitors are able to show only a relatively few tone graduations. Although the Applix provides up to 16 shades, not all will be visible on most monochrome monitors.

As the Applix video is programmable, a Memory Resident Driver (see the *Technical Reference Manual* for details) can alter the scan frequencies, and the number of pixels on the screen. It is thus possible to change the display from 640 by 200 pixels, up to 640 by 350 pixels. This requires an increased horizontal scan frequency, and thus a different style of display. A suitable frequency turns out to be closely equivalent to that required by the (non TV standard) IBM Monochrome or Hercules monitor. Thus we recommend a 'dual scan' monitor for monochrome use.

The 640 by 350 display (devised by Conal Walsh, who also did the ZRDOS port) is capable of much better text and graphics displays, and is highly recommended. However, in colour, it requires a much more expensive, much higher quality, Multisync monitor. This option gives you the equivalent of IBM EGA output. Prices of these range from round $400 (some of pretty terrible quality) to over $1000 (some of which are very nice indeed). There is also a single chip modification required in your Applix, although some monitors will run without this change.

Finally, a $30 modification to your Applix allows you to run Stephen A Uhler's MGR windowing system, ported from Sun Unix workstations, providing multiple, resizable windows and mouse support, on a 940 by 512 display. This requires a Multisync monitor.

Ask Applix or the User Group for advice, before you buy a monitor, if this discussion of the differences between monitors is unclear to you.

The I/O ports

Data going to and from the 1616's serial ports is buffered and interrupt driven. The default buffer size is 200 bytes for input (per channel) and 200 bytes for output. This simply means that a certain amount of communication can take place even when the computer is busy doing other things. See the documentation for the inbuilt 'serial' command for further description of the serial port software. You don't actually need to know about this, unless you want to use the serial RS232 ports.

Data transmission to the parallel printer is also interrupt driven and buffered. The default buffer size is 64 bytes. As with the keyboard buffer, the size of this buffer can be altered by advanced users by means of the ***new_cbuf*** system call.

]

Index

Table of Contents